João M. Fernandes
Ralf Lämmel
Joost Visser
João Saraiva (Eds.)

Tutorial

LNCS 6491

Generative and Transformational Techniques in Software Engineering III

International Summer School, GTTSE 2009
Braga, Portugal, July 2009
Revised Papers

Springer

Lecture Notes in Computer Science 6491

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

João M. Fernandes Ralf Lämmel
Joost Visser João Saraiva (Eds.)

Generative and Transformational Techniques in Software Engineering III

International Summer School, GTTSE 2009
Braga, Portugal, July 6-11, 2009
Revised Papers

Volume Editors

João M. Fernandes
João Saraiva
Universidade do Minho
Departamento de Informática
Campus de Gualtar, 4710-057 Braga, Portugal
E-mail: {jmf,jas}@di.uminho.pt

Ralf Lämmel
Universität Koblenz-Landau
FB 4, Institut für Informatik
B127, Universitätsstraße 1, 56070 Koblenz, Germany
E-mail: rlaemmel@gmail.com

Joost Visser
Software Improvement Group
A.J. Ernststraat 595-H, 1082 LD Amsterdam, The Netherlands
E-mail: j.visser@sig.nl

Library of Congress Control Number: 2010941367

CR Subject Classification (1998): D.2, D.3, F.3, D.1, F.4.2, D.2.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-18022-1 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-18022-4 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

The third instance of the international summer school on Generative and Transformational Techniques in Software Engineering (GTTSE 2009) was held in Braga, Portugal, July 6–11, 2009. In this volume, you find revised and extended lecture notes for most of the long and short summer-school tutorials as well as a small number of peer-reviewed papers that originated from the participants' workshop.

The mission of the GTTSE summer school series is to bring together PhD students, lecturers, as well as other researchers and practitioners who are interested in the generation and the transformation of programs, data, models, metamodels, documentation, and entire software systems. This mission crosscuts many areas of software engineering, e.g., software reverse and re-engineering, model-driven engineering, automated software engineering, generic language technology, software language engineering—to name a few. These areas differ in interesting ways, for example, with regard to the specific sorts of metamodels (or grammars, schemas, formats, etc.) that underlie the involved artifacts, and with regard to the specific techniques that are employed for the generation and the transformation of the artifacts.

The first two instances of the school were held in 2005 and 2007, and their post-proceedings appeared as volumes 4143 and 5235 in Springer's *LNCS* series.

The 2009 instance of GTTSE offered eight long tutorials, given by renowned representatives of complementary approaches and problem domains. Each tutorial combined foundations, methods, examples, and tool support. The program of the summer school featured another six short(er) tutorials, which presented more specific contributions to generative and transformational techniques. All tutorial presentations were invited by the organizers to complement each other in terms of the chosen application domains, case studies, and the underlying concepts. Yet another module in the program was a Research 2.0 event which combined tutorial-like aspects with a great discussion.

The program of the school also included a participants' workshop to which all students had been asked to submit an extended abstract beforehand. The Organizing Committee reviewed these extended abstracts and invited ten students to present their work at the workshop. The quality of this workshop was exceptional, and two awards were granted by a board of senior researchers that was formed at the school.

The program of the school remains available online.[1]

This volume contains revised and extended lecture notes for most of the long and short summer-school tutorials as well as a small number of peer-reviewed

[1] http://gttse.wikidot.com/2009

papers that originated from the participants' workshop. Each of the included seven long tutorial papers was reviewed by two members of the Scientific Committee of GTTSE 2009. Each of the included six short tutorial papers was reviewed by three members. The tutorial papers were primarily reviewed to help the authors with compiling original, readable and useful lecture notes. The three included participant contributions were peer-reviewed with three reviews per paper. For all papers, two rounds of reviewing and revision were executed.

We are grateful to our sponsors for their support and to all lecturers and participants of the school for their enthusiasm and hard work in preparing excellent material for the school itself and for these proceedings. Thanks to their efforts the event was a great success, which we trust the reader finds reflected in this volume. Our gratitude is also due to all members of the scientific committee who not only helped with the labor-intensive review process that substantially improved all contributions, but also sent their most suitable PhD students to the school.

The next edition of GTTSE, GTTSE 2011, will be organized in Braga again, and it will be co-located with the 4th International Conference on Software Language Engineering. This co-location will provide for excellent synergies.

October 2010

João M. Fernandes
Ralf Lämmel
João Saraiva
Joost Visser

Organization

GTTSE 2009 was hosted by the Departamento de Informática, Universidade do Minho, Braga, Portugal.

Executive Committee

Program Co-chairs

João M. Fernandes	Universidade do Minho, Braga, Portugal
Ralf Lämmel	Universität Koblenz-Landau, Germany

Organizing Co-chairs

João Saraiva	Universidade do Minho, Braga, Portugal
Joost Visser	Software Improvement Group, Amsterdam, The Netherlands

Scientific Committee

Jean Bézivin	Université de Nantes, France
Charles Consel	LaBRI/INRIA, France
Erik Ernst	Aarhus University, Denmark
João M. Fernandes	Universidade do Minho, Portugal
Lidia Fuentes	University of Málaga, Spain
Jeff Gray	University of Alabama at Birmingham, USA
Reiko Heckel	University of Leicester, UK
Zhenjiang Hu	National Institute of Informatics, Japan
Ralf Lämmel	Universität Koblenz-Landau, Germany
Juan de Lara	Universidad Autónoma de Madrid, Spain
Julia Lawall	University of Copenhagen, Denmark
Johan Lilius	Åbo Akademi University, Finland
Antónia Lopes	Universidade de Lisboa, Portugal
Marjan Mernik	University of Maribor, Slovenia
José N. Oliveira	Universidade do Minho, Portugal
Richard Paige	University of York, UK
Zoltán Porkoláb	University Eötvös Loránd, Hungary
Andreas Prinz	University of Agder, Norway
Markus Pueschel	CMU, USA
Awais Rashid	Lancaster University, UK
Andy Schürr	Technical University Darmstadt, Germany
Sérgio Soares	Universidade de Pernambuco, Brazil
Peter Thiemann	Universität Freiburg, Germany
Eelco Visser	Delft University of Technology, The Netherlands
Albert Zündorf	University of Kassel, Germany

Organizing Committee

João Paulo Fernandes	Universidade do Minho, Braga, Portugal
Ralf Lämmel	Universität Koblenz-Landau, Germany
João Saraiva	Universidade do Minho, Braga, Portugal
Joost Visser	Software Improvement Group, Amsterdam, The Netherlands
Vadim Zaytsev	Universität Koblenz-Landau, Germany

Sponsoring Institutions

Universität Koblenz-Landau
Departamento de Informática, Universidade do Minho
Centro de Ciências e Tecnologias de Computação
Fundação para a Ciência e a Tecnologia
Luso-American Foundation
Software Improvement Group
Efacec
Multicert

Table of Contents

Part I – Long Tutorials

Part II – Short Tutorials

Part III – Participants Contributions

An Introduction to Software Product Line Refactoring

Paulo Borba

Informatics Center
Federal University of Pernambuco
phmb@cin.ufpe.br

Abstract. Although software product lines (PLs) can bring significant productivity and quality improvements through strategic reuse, bootstrapping existing products into a PL, and extending a PL with more products, is often risky and expensive. These kinds of PL derivation and evolution might require substantial effort and can easily affect the behavior of existing products. To reduce these problems, we propose a notion of product line refactoring and associated transformation templates that should be part of a PL refactoring catalogue. We discuss how the notion guides and improves safety of the PL derivation and evolution processes; the transformation templates, particularly when automated, reduce the effort needed to perform these processes.

1 Introduction

A software product line (PL) is a set of related software products that are generated from reusable assets. Products are related in the sense that they share common functionality. Assets correspond to components, classes, property files, and other artifacts that are composed in different ways to specify or build the different products. For example, in the simplified version of the Rain of Fire mobile game product line shown in Fig. 1, we have three products varying only in how they support clouds in the game background, as this impacts on product size and therefore demands specific implementations conforming to different mobile phones' memory resources. The classes and images are reused by the three products; the clouds image, for instance, is used by two products. Each XML file and aspect [KHH+01] (`.aj` file), which are common variability implementation mechanisms [GA01, AJC+05], specify cloud specific data and behavior, so are only reused when considering other products not illustrated in Fig. 1.

This kind of reuse targeted at a specific set of products can bring significant productivity and time to market improvements [PBvdL05, Chapter 21][vdLSR07, Chapters 9-16]. The extension of a PL with a new product demands less effort because existing assets can likely make up to a large part of the new product. The maintenance of existing products also demands less effort because changes in a single asset often have an impact on more than one product. Indirectly, we can improve quality too [vdLSR07], since assets are typically more exposed and tested through their use in different products.

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 1–26, 2011.

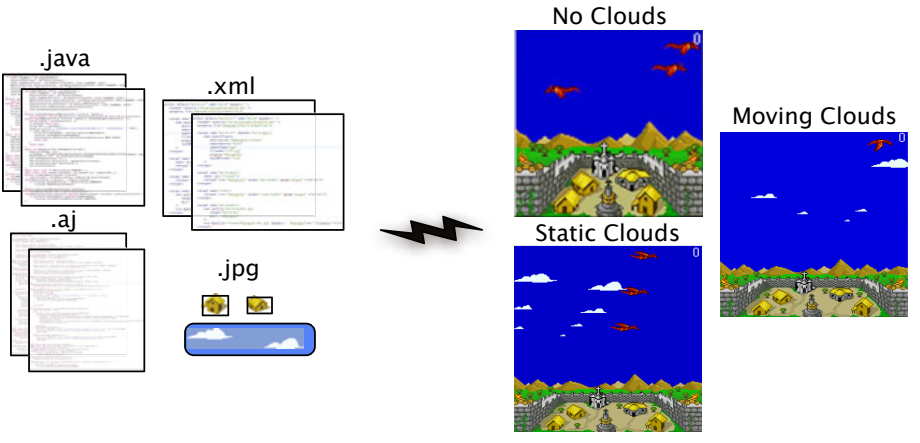


Fig. 1. Mobile game product line

To obtain these benefits with reduced upfront investment, previous work proposes to minimize the initial product line analysis and development process by bootstraping existing related products into a PL [Kru02, CN01, AJC+05]. Through an extraction process, we can separate variations from common parts, and then discard duplicate common parts. In Fig. 2, the first pair of arrows and the subsequent arrow illustrate this process for the code[1] of two related products: the same mobile game with moving clouds, differing only in their image loading policy. A similar process applies for evolving a PL, when adding new products might require extracting a part shared by existing products but not appropriate for some new products. In this case, a previously common part becomes a variation in the new PL. For example, the rightmost arrow in Fig. 2 shows that we extracted the code associated to clouds to a new aspect, which is an adequate mechanism to separate the cloud variations from the other parts of the game. We can now, by not including the Clouds aspect, have games without clouds in the product line.

Although useful to reduce upfront investment and the risk of losing it due to project failure, this extraction process might be tedious. Manually extracting and changing different parts of the code requires substantial effort, especially for analyzing the conditions that ensure the correctness of the extraction process. In fact, this process can easily introduce defects, modifying the behavior exhibited by products before the extraction process, and compromising the promised benefits on other dimensions of costs and risks.

To minimize these problems, the proposed extraction process could benefit from automatic refactorings: behavior-preserving source-to-source transformations that improve some quality factor, usually reusability and maintainability [Rob99, Fow99]. Refactorings automate tedious changes and analyses,

[1] The same process applies for other artifacts such as requirements documents, test cases, and design models [TBD06].

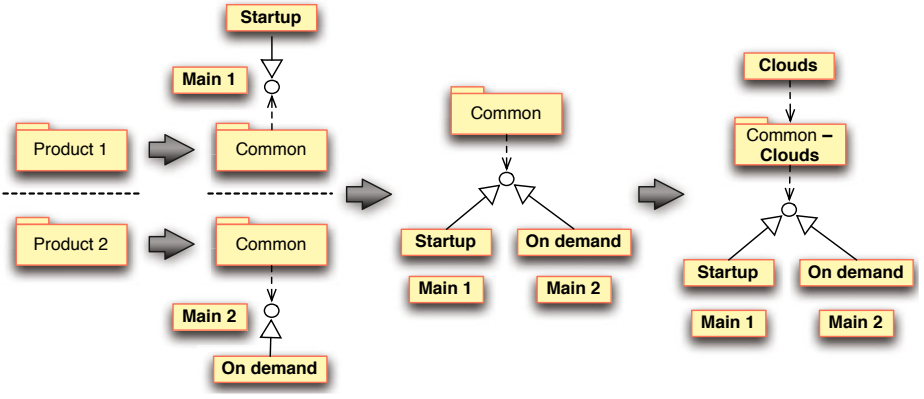


Fig. 2. Product line extraction

consequently reducing costs and risks.[2] They can help too by providing guidance on how to structure extracted variants. All that could be useful for deriving a PL from existing products, and also for evolving a PL by simply improving its design or by adding new products while preserving existing ones.

However, existing refactoring notions [Opd92, Fow99, BSCC04, CB05] focus on transforming single programs, not product lines (PLs). These notions, therefore, might justify the pair of transformations in the first step of Fig. 2, for independently transforming two programs [TBD06, KAB07, AJC+05, AGM+06]. But they are not able to justify the other two steps. The second requires merging two programs into a product line; the resulting product line has conflicting assets such as the two main classes, so it does not actually correspond to a valid program. Similarly, the third step involves transforming conflicting assets, which again are not valid programs.

Similar limitations apply for refactoring of other artifacts, including design models [GMB05, MGB08]. These notions focus on refactoring a single product, not a product line. Besides that, as explained later, in a product line we typically need extra artifacts, such as feature models [KCH+90, CE00], for automatically generating products from assets. So a PL refactoring notion should overcome the single product limitations just mentioned and go beyond reusable assets, transforming the extra artifacts as well.

We organize this text as follows. Section 2 introduces basic concepts and notation for feature models and other extra product line artifacts [CE00, BB09]. This section also emphasizes informal definitions for the semantics of these artifacts, and for a notion of program refinement [SB04, BSCC04], with the aim of explicitly introducing notation and the associated intuitions. This aligns with our focus on establishing concepts about an emerging topic, rather than providing a complete formalization or reporting practical experience on product line

[2] Current refactoring tools might still introduce defects because they do not fully implement some analyses [ST09].

refactoring [ACV+05, AJC+05, AGM+06, TBD06, CBS+07, KAB07, ACN+08]. Following that, in Sec. 3, we propose and formalize a notion of product line refactoring. This goes beyond refactoring of feature models, which is the focus of our previous work [AGM+06, GMB08]. The notion and the formalization are independent of languages used, for example, to describe feature models and code assets. They depend only on the interfaces expressed by the informal definitions that appear in Sec. 2; that is why the theorems in Sec. 3 do not rely, for example, on a formalization of feature models. To illustrate one of the main applications of such a refactoring notion, Sec. 4 shows different kinds of transformation templates that can be useful for refactoring product lines. This goes beyond templates for transforming feature models [AGM+06, GMB08], considering also other artifacts, both in isolation and in an integrated way. As the templates use specific notation for feature models and the other artifacts, proving that the transformations are sound with respect to the refactoring notion requires the formal semantics of the used notations. However, as our main aim is to stimulate the derivation of comprehensive refactoring catalogues, considering different notations and semantic formalizations for product line artifacts [CHE05, Bat05, SHTB07, GMB08], we prefer not to lose focus by introducing details of the specific notations adopted here.

2 Software Product Line Concepts

In the PL approach adopted in this text, Feature Models (FMs) and Configuration Knowledge (CK) [CE00] enable the automatic generation of products from assets. A FM specifies common and variant features among products, so we can use it to describe and select products based on the features they support. A CK relates features and assets, specifying which assets implement possible feature combinations. Hence we use a CK to actually build a product given chosen features for that product. We now explain in more detail these two kinds of artifacts, using examples from the Mobile Media product line [FCS+08], which contains applications – such as the one illustrated in Fig. 3 – that manipulate photos, music, and video on mobile devices.

2.1 Feature Models

A feature model is essentially represented as a tree, containing features and information about how they are related. Features basically abstract groups of associated requirements, both functional and non-functional. Relationships between a parent feature and its child features (subfeatures) indicate whether the subfeatures are *optional* (present in some products but not in others, represented by an unfilled circle), *mandatory* (present in all products, represented by a filled circle), *or* (every product has at least one of them, represented by a filled triangular shape), or *alternative* (every product has exactly one of them, represented by an unfilled triangular shape). For example, Fig. 4 depicts a simplified Mobile Media FM, where Sorting is optional, Media is mandatory, Photo and Music are or-features, and the two illustrated screen sizes are alternative.

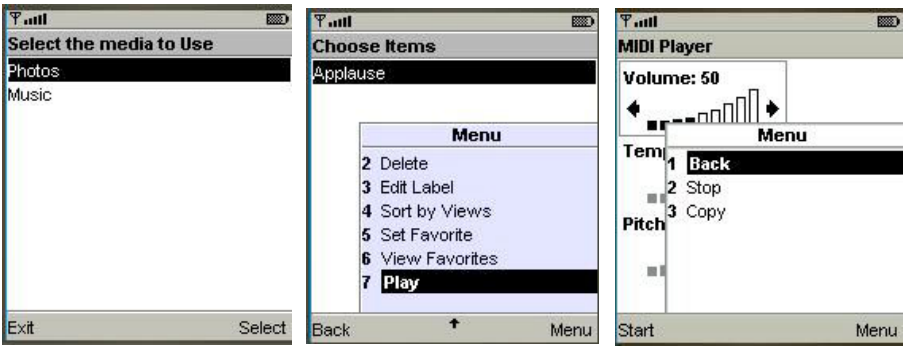


Fig. 3. Mobile Media screenshots

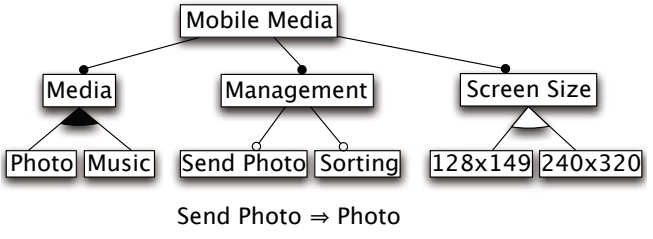


Fig. 4. Mobile Media simplified feature model

Besides these relationships, feature models may contain propositional logic formulas about features. We use feature names as atoms to indicate that a feature should be selected. So negation of a feature indicates that it should not be selected. For instance, the formula just below the tree in Fig. 4 states that feature Photo must be present in some product whenever we select feature Send Photo. So {Photo, Send Photo, 240x320}, together with the mandatory features, which hereafter we omit for brevity, is a valid feature selection (product configuration), but {Music, Send Photo, 240x320} is not. Likewise {Music, Photo, 240x320} is a possible configuration, but {Music, Photo, 240x320, 128x149} is not because it breaks the Screen Size alternative constraint. In summary, a valid configuration is one that satisfies all FM constraints, specified both graphically and through formulas. Each valid configuration corresponds to one PL product, expressed in terms of the features it supports. So the following definition captures the intuition that a FM denotes the set of products in a PL.

Definition 1 ⟨FM semantics⟩
The semantics of a feature model F, represented as $[\![F]\!]$, is the set of all valid product configurations (sets of feature names) of F.

□

To introduce the notion of PL refactoring, discuss refactorings of specific PLs, and even derive general theorems about this notion, this is all we need to know

about FMs. So we omit here the formal definitions of $[\![F]\!]$ and valid configurations, which appear elsewhere [GMB08, AGM+06] for the notation used in this section. There are alternative FM notations [CHE05, Bat05, SHTB07, GMB08], but, as shall be clear later, our notion of PL refactoring does not depend on the used FM notation. Our notion works for any FM notation whose semantics can be expressed as a set of configurations, as explicitly captured by Definition 1. On the other hand, to prove soundness of refactoring transformation templates, as showed later, we would need a formal definition of $[\![F]\!]$ because the templates use a specific FM notation. But this is out of the scope of this work, which aims to establish PL refactoring concepts and stimulate further work – such as the derivation of refactoring catalogues considering different notations and semantic formalizations for PL artifacts – rather than to develop a theory restricted to a specific FM notation.

2.2 Configuration Knowledge

As discussed in the previous section, features are groups of requirements, so they must be related to the assets that realize them. Abstracting some details of previous work [BB09], a CK is a relation from feature expressions (propositional formulas having feature names as atoms) to sets of asset names. For example, showing the relation in tabular form, the following CK

Mobile Media	MM.java, ...
Photo	Photo.java, ...
Music	Music.java, ...
Photo ∨ Music	Common.aj, ...
Photo ∧ Music	AppMenu.aj, ...
⋮	⋮

establishes that if the Photo and Music features are both selected then the `AppMenu` aspect, among other assets omitted in the fifth row, should be part of the final product. Essentially, this PL uses the `AppMenu` aspect as a variability implementation mechanism [GA01, AJC+05] that has the effect of presenting the left screenshot in Fig. 3.[3] For usability issues, this screen should not appear in products that have only one of the Media features. This is precisely what the fifth row, in the simplified Mobile Media CK, specifies. Similarly, the Photo and Music implementations share some assets, so we write the fourth row to avoid repeating the asset names on the second and third rows.

Given a valid product configuration, the evaluation of a CK yields the names of the assets needed to build the corresponding product. In our example, the configuration {Photo, 240x320}[4] leads to

{`MM.java`, ..., `Photo.java`, ..., `Commom.aj`, ...}.

[3] We could use other variability implementation mechanisms, but that is not really needed to explain the CK concept, nor our refactoring notion, which does not depend on the type of assets used by PLs.

[4] Remember we omit mandatory features for brevity.

This gives the basic intuition for defining the semantics of a CK.

Definition 2 ⟨CK semantics⟩
The semantics of a configuration knowledge K, represented as $[\![K]\!]$, is a function that maps product configurations into sets of asset names, in such a way that, for a configuration c, an asset name a is in the set $[\![K]\!]c$ iff there is a row in K that contains a and its expression evaluates to true according to c. □

We again omit the full formalization since, as discussed before, it is only necessary to formally prove soundness of PL refactoring transformation templates that use this particular notation for specifying the CK. Our notion of PL refactoring, and associated properties, work for any CK notation whose semantics can be expressed as a function that maps configurations into sets of assets names. This is why we emphasize the $[\![K]\!]$ notation in Definition 2, and use it later when defining PL refactoring.

2.3 Assets

Although the CK in the previous section refers only to code assets, in general we could also refer to requirements documents, design models, test cases, image files, XML files, and so on. For simplicity, here we focus on code assets as they are equivalent to other kinds of assets with respect to our interest in PL refactoring. The important issue here is not the nature of the asset contents, but how we compare assets and refer to them in the CK. We first discuss the asset reference issue.

Asset mapping. With respect to CK references, remember that we might have conflicting assets in a PL. For instance, on the right end of Fig. 2, we have two `Main` classes. They have the same name and differ only on how they instantiate their image loading policy:

```
class Main {                          class Main {
    ...new StartUp(...);...               ...new OnDemand(...);...
}                                     }
```

We could avoid the duplicate class names by having a single `Main` class that reads a configuration file and then decides which policy to use, but we would then need two configuration files with the same name. Duplicate names might also apply to other classes and assets in a PL, so references to those names would also be ambiguous.

To avoid the problem, we assume that the names that appear in a CK might not exactly correspond to the names used in asset declarations. For instance, the names "Main 1" and "On demand" in this CK

⋮	⋮
On Demand	Main 2, On demand
Start Up	Main 1, Startup
On Demand ∨ Start Up	Common.java
⋮	⋮

are not really class names. Instead, the PL keeps a mapping such as the one in Fig. 5, from the CK names to actual assets. So, besides a FM and a CK, a PL actually contains an asset mapping, which basically corresponds to an environment of asset declarations.

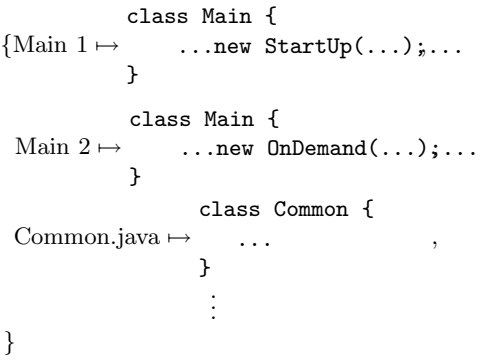


Fig. 5. Asset mapping

Asset refinement. Finally, for defining PL refactoring, we must introduce a means of comparing assets with respect to behavior preservation. As we focus on code, we use existing refinement definitions for sequential programs [SB04, CB05].

Definition 3 ⟨Program refinement⟩
For programs p_1 and p_2, p_1 is refined by p_2, denoted by

$$p_1 \sqsubseteq p_2$$

when p_2 is at least as good as p_1 in the sense that it will meet every purpose and satisfy every input-output specification satisfied by p_1. We say that p_2 preserves the (observable) behavior of p_1.

□

Refinement relations are pre-orders: they are reflexive and transitive. They often are partial-orders, being anti-symmetric too, but we do not require that for the just introduced relation nor for the others discussed in the remaining of the text.

For object-oriented programs, we have to deal with class declarations and method calls, which are inherently context-dependent; for example, to understand the meaning of a class declaration we must understand the meaning of its superclass. So, to address context issues, we make declarations explicit when dealing with object-oriented programs: '$cds \bullet m$' represents a program formed by a set of class declarations cds and a command m, which corresponds to the `main` method in Java like languages. We can then express refinement of class declarations as program refinement. In the following, juxtaposition of sets of class declarations represents their union.

Definition 4 ⟨Class declaration refinement⟩
For sets of class declarations cds_1 and cds_2, cds_1 is refined by cds_2, denoted by

$$cds_1 \sqsubseteq_{cds,m} cds_2$$

in a context cds of "auxiliary" class declarations for cds_1 and cds_2, and a main command m, when

$$cds_1\ cds \bullet m \sqsubseteq cds_2\ cds \bullet m.$$

□

For asserting class declaration refinement independently of context, we have to prove refinement for arbitrary cds and m that form valid programs when separately combined with cds_1 and cds_2 [SB04].

This definition captures the notion of behavior preservation for classes, so, using an abstract programming notation [BSCC04, SB04], code transformations of typical object-oriented refactorings can be expressed as refinements. For example, the following equivalence (refinement in both directions) establishes that we can move a public attribute a from a class C to a superclass B, and vice-versa.

class B **extends** A
  ads
  ops
end
class C **extends** B
  pub $a : T; ads'$
  ops'
end
	$=_{cds,m}$	**class** B **extends** A
		  pub $a : T; ads$
		  ops
		end
		class C **extends** B
		  ads'
		  ops'
		end

To move the attribute up to B, it is required that this does not generate a name conflict: no subclass of B, other than C, can declare an attribute with the same name, to avoid attribute redefinition or hiding. We can move a from B to C provided that a is used only as if it were declared in C. The formal pre-conditions, and a comprehensive set of similar transformations, appear elsewhere [BSCC04, SB04]. We omit them here because they focus on a specific programming language, whereas we want to establish a notion of PL refactoring that is language independent, as long as the language has a notion of program refinement such as the one just discussed. Even the overall PL refactoring templates that we show later depend on specific FM and CK languages but are programming language independent.

We could also have a similar definition of refinement for aspects and other code artifacts, but as they can all be expressed in terms of program refinement, hereafter we use the fundamental program refinement relation $\sqsubseteq$, and assume translations, for example of a set of classes with a specific main class, into the formats discussed in this section. In summary, such a reflexive and transitive notion of program refinement is all we need for our notion of PL refactoring, its basic properties, and the overall PL templates we discuss later. For reasoning about reusable assets in isolation we also need a notion of asset refinement

(such as the one for class declaration). This should be compositional, in the sense that refining an asset that is part of a given program implies refinement of the whole program.

2.4 Product Lines

We can now provide a precise definition for product lines, and a better account than what we represent in Fig. 1, which illustrates only assets and products. In particular, we make sure the three PL elements discussed in the previous sections are consistent in the sense of referring only to themselves. We also require each PL product to be a valid program[5] in its target languages.

Definition 5 ⟨Product line⟩
For a feature model F, an asset mapping A, and a configuration knowledge K, we say that tuple

$$(F, A, K)$$

is a product line when the expressions in K refer only to features in F, the asset names in K refer only to the domain of A, and, for all $c \in [\![F]\!]$,

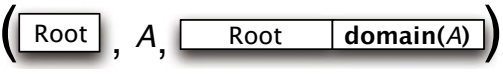
$$A\langle [\![K]\!]c \rangle$$

is a valid program, where $A\langle S \rangle$, for a mapping A and a set S, is an abbreviation for $\{A(s) \mid s \in S\}$.

□

For object-oriented languages, $A\langle [\![K]\!]c \rangle$ should be a well-typed set of classes containing a single main class, with a main method. This validity constraint in the definition is necessary because missing an entry on a CK might lead to products that are missing some parts and are thus invalid. Similarly, a mistake when writing a CK or asset mapping entry might yield an invalid program due to conflicting assets, like two aspects that we use as variability mechanism and introduce methods with the same signature in the same class. Here we demand CKs to be correct as explained.

It is useful to note that, following this definition, we can see a single system as a single-product PL:



The feature model contains a single feature Root and no constraints, the asset mapping A simply maps declared names to the corresponding system asset declarations, and the CK contains a single entry relating Root to all asset names. For a product line in this form, the corresponding single system is simply the image of A.

[5] Remember our focus on code artifacts. In general, we should require products to contain valid artifacts no matter the languages and notations (modeling, scripting, programming, etc.) used to describe them.

3 Product Line Refactoring

Now that we better understand what a PL is, we can define a notion of PL refactoring to address the issues mentioned in Sec. 1. Similar to program refactorings, PL refactorings are behavior-preserving transformations that improve some quality factor. However, they go beyond source code, and might transform FMs and CKs as well. We illustrate this in Fig. 6, where we refactor the simplified Mobile Media product line by renaming the feature Music. As indicated by check marks, this renaming requires changing the FM, CK, and asset mapping; due to a class name change, we must apply a global renaming, so the main method and other classes beyond `Music.java` chang too.

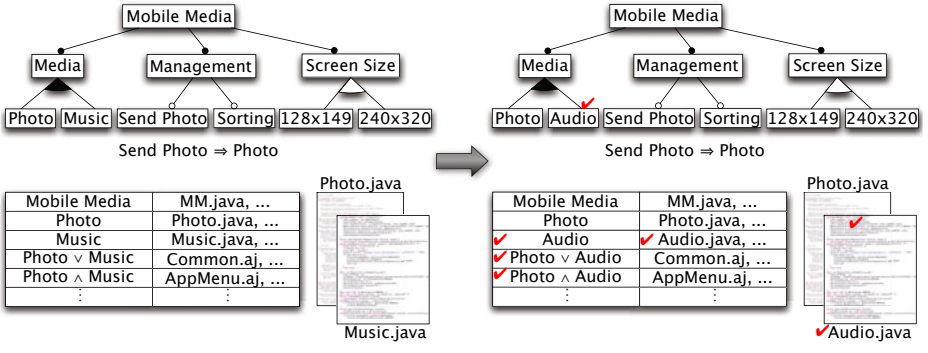


Fig. 6. PL renaming refactoring

The notion of behavior preservation should be also lifted from programs to product lines. In a PL refactoring, the resulting PL should be able to generate products (programs) that behaviorally match the original PL products. So users of an original product cannot observe behavior differences when using the corresponding product of the new PL. With the renaming refactoring, for example, we have only improved the PL design: the resulting PL generates a set of products exactly equivalent to the original set. But it should not be always like that. We consider that the better product line might generate more products than the original one. As long as it generates enough products to match the original PL, users have no reason to complain. For instance, by adding the optional Copy feature (see Fig. 7), we refactor our example PL. The new PL generates twice as many products as the original one, but half of them – the ones that do not have feature Copy – behave exactly as the original products. This ensures that the transformation is safe; we extended the PL without impacting existing users.

3.1 Formalization

We formalize these ideas as a notion of refinement for product lines, defined in terms of program refinement (see Definition 3). Each program generated by the original PL must be refined by some program of the new, improved, PL.

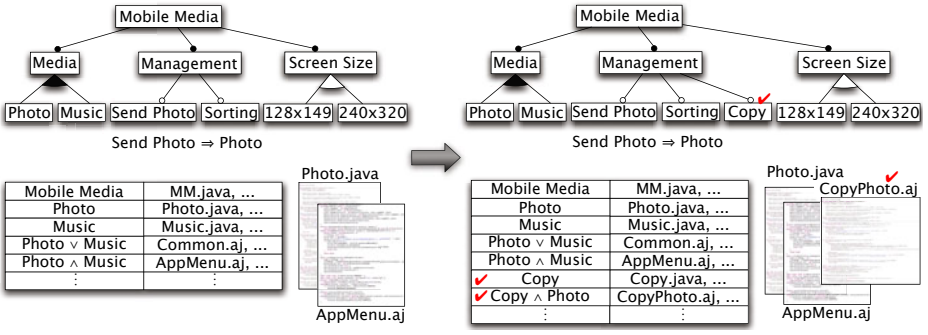


Fig. 7. Adding an optional feature refactoring

Definition 6 ⟨PL refinement⟩
For product lines (F, A, K) and (F', A', K'), the first is refined by the second, denoted

$$(F, A, K) \sqsubseteq (F', A', K')$$

whenever

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F']\!] \cdot A\langle [\![K]\!]c\rangle \sqsubseteq A'\langle [\![K']\!]c'\rangle$$

□

Remember that, for a configuration c, configuration knowledge K, and asset mapping A related to a given PL, $A\langle [\![K]\!]c\rangle$ is a set of assets that constitutes a valid program. So $A\langle [\![K]\!]c\rangle \sqsubseteq A'\langle [\![K']\!]c'\rangle$ refers to the program refinement notion discussed in Sec. 2.3.

Now, for defining PL refactoring, we need to capture quality improvement as well.

Definition 7 ⟨PL refactoring⟩
For product lines PL and PL', the first is refactored by the second, denoted

$$PL \lll PL'$$

whenever

$$PL \sqsubseteq PL'$$

and PL' is better than PL with respect to quality factors such as reusability and maintainability [Rob99, Fow99].

□

We provide no formalization of quality improvement, as this is subjective and context dependent. Nevertheless, this definition formalizes to a good extent, and slightly generalizes, our previous definition [AGM+06]: "a PL refactoring is a change made to the structure of a PL in order to improve (maintain or increase) its configurability, make it easier to understand, and cheaper to modify without changing the observable behavior of its original products". The difficulty of formally capturing quality improvement is what motivates the separate notions of refactoring and refinement. We can only be formal about the latter.

3.2 Examples and Considerations

To explore the definitions just introduced, let us analyze concrete PL transformation scenarios.

Feature names do not matter. First let us see how the definitions apply to the transformation depicted by Fig. 6. The FMs differ only by the name of a single feature. So they generate the same set of configurations, modulo renaming. For instance, for the source (left) PL configuration {Music, 240x320} we have the target (right) PL configuration {Audio, 240x320}. As the CKs have the same structure, evaluating them with these configurations yield

{`Commmon.aj`, `Music.java`, ...}

and

{`Commmon.aj`, `Audio.java`, ...}.

The resulting sets of asset names differ at most by a single element: `Audio.java` replacing `Music.java`. Finally, when applying these sets of names to both asset mappings, we obtain the same assets modulo global renaming, which is a well known refactoring for closed programs. This implies behavior-preservation and therefore program refinement, which is precisely what, by Definition 6, we need for assuring that the source PL is refined by the target PL. Refactoring, by Definition 7, follows from the fact that Audio is a more representative name for what is actually manipulated by the applications.

This example shows that our definitions focus on the PLs themselves, that is, the sets of generated products. Contrasting with our previous notion of feature model refactoring [AGM+06], feature names do not matter. So users will not notice they are using products from the new PL, although PL developers might have to change their feature nomenclature when specifying product configurations. Not caring about feature names is essential for supporting useful refactorings such as the just illustrated feature renaming and others that we discuss later.

Safety for existing users only. To further explore the definitions, let us consider now the transformation shown in Fig. 7. The target FM has an extra optional feature. So it generates all configurations of the source FM plus extensions of these configurations with feature Copy. For example, it generates both {Music, 240x320} and {Music, 240x320, Copy}. For checking refinement, we focus only on the configurations common to both FMs – configurations without Copy. As the target CK is an extension of the source CK for dealing with cases when Copy is selected, evaluating the target CK with any configuration without Copy yields the same asset names yielded by the source CK with the same configuration. In this restricted name domain, both asset mappings are equal, since the target mapping is an extension of the first for names such as `CopyPhoto.java`, which appears only when we select Copy. Therefore, the resulting assets produced by each PL are the same, trivially implying program refinement and then PL

refinement. Refactoring follows because the new PL offers more reuse opportunities due to new classes and aspects such as `CopyPhoto.java`.

By focusing on the common configurations to both FMs, we check nothing about the new products offered by the new PL. In fact, they might even not operate at all. Our refactoring notion assures only that users of existing products will not be disappointed by the corresponding products generated by the new PL. We give no guarantee to users of the new products, like the ones with Copy functionalities in our example. So refactorings are safe transformations only in the sense that we can change a PL without impacting existing users.

Non refactorings. As discussed, the transformation depicted in Fig. 6 is a refactoring. We transform classes and aspects through a global renaming, which preserves behavior for closed programs. But suppose that, besides renaming, we change the `AppMenu.aj`[6] aspect so that, instead of the menu on the left screenshot in Fig. 3, we have a menu with "Photos" and "Audio" options. The input-output behavior of new and original products would then not match, and users would observe the difference. So we would not be able to prove program refinement, nor PL refinement and refactoring, consequently.

Despite not being a refinement, this menu change is an useful PL improvement, and should be carried on. The intention, however, is to change behavior, so developers will not be able to rely on the benefits of checking refinement and refactoring. They will have to test the PL to make sure the effect of the applied transformations actually corresponds to the expected behavior changes. The benefits of checking for refactoring only apply when the intention of the transformation is to improve PL configurability or internal structure, without changing observable behavior.

Similarly, adding a mandatory feature such as Logging to a PL is not a refactoring. This new feature might be an important improvement for the PL, but it deliberately changes the behavior of all products, so we cannot rely on refactoring to assure that the transformation is safe. In fact, by checking for refactoring, we learn that the transformation is not safe, as expected. If the intention was to preserve behavior, a failed refactoring check would indicate a problem in the transformation process.

Figure 8 shows another possibly useful transformation that is not a refactoring. Suppose that, for market reasons, we removed the 128x149 screen size alternative feature, so the target PL generates only half of the products generated by the source PL. Configurations such as {Music, 240x320}, without the removed screen size, yield exactly the same product in both PLs. Behavior is preserved for these configurations because the asset mappings and CKs differ only on entries related to `SS1.aj`. However, refactoring does not hold because of configurations such as {Music, 128x149}. The products associated with these configurations cannot be matched by products from the target PL, which is required for PL refinement.

[6] See Sec. 2.2 for understanding the role this aspect plays.

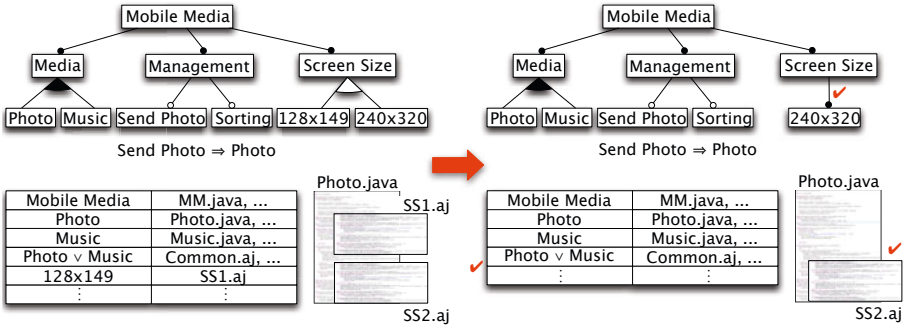


Fig. 8. Non refactoring: removing an alternative feature

3.3 Special Cases

For checking refinement as in the previous section, it is useful to know PL refinement properties that hold for special cases. First, when PLs differ only by their CK, we can check refinement by checking if the CKs differ only syntactically.

Theorem 1 ⟨Refinement with different CKs⟩
For product lines (F, A, K) and (F, A, K'), if

$$[\![K]\!] = [\![K']\!]$$

then

$$(F, A, K) \sqsubseteq (F, A, K')$$

Proof: First assume that $[\![K]\!] = [\![K']\!]$. By Definition 6, we have to prove that

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F]\!] \cdot A\langle [\![K]\!]c\rangle \sqsubseteq A\langle [\![K']\!]c'\rangle$$

From our assumption, this is equivalent to

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F]\!] \cdot A\langle [\![K]\!]c\rangle \sqsubseteq A\langle [\![K]\!]c'\rangle$$

For an arbitrary $c \in [\![F]\!]$, just let c' be c and the proof follows from program refinement reflexivity.

□

Note that the reverse does not hold because the asset names generated by K and K' might differ for assets that have no impact on product behavior,[7] or for assets that have equivalent behavior but have different names in the PLs.

We can also simplify checking when only asset mappings are equal. In this case we still bypass program refinement checking, which is often difficult.

[7] Obviously an anomaly, but still possible.

Theorem 2 ⟨Refinement with equal asset mappings⟩
For product lines (F, A, K) and (F', A, K'), if

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F']\!] \cdot [\![K]\!]c = [\![K']\!]c'$$

then

$$(F, A, K) \sqsubseteq (F', A, K')$$

Proof: First assume that $\forall c \in [\![F]\!] \cdot \exists c' \in [\![F']\!] \cdot [\![K]\!]c = [\![K']\!]c'$. By Definition 6, we have to prove that

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F']\!] \cdot A\langle [\![K]\!]c\rangle \sqsubseteq A\langle [\![K']\!]c'\rangle$$

For an arbitrary $c \in [\![F]\!]$, our assumption gives us a $c'_1 \in [\![F']\!]$ such that

$$[\![K]\!]c = [\![K']\!]c'_1$$

Just let c' be c'_1 and the proof follows from this equality, equational reasoning, and program refinement reflexivity.

□

Again, the reverse does not hold, for similar reasons. Noting that, in the first theorem, '$[\![K]\!] = [\![K']\!]$' actually amounts to '$\forall c \in [\![F]\!] \cdot [\![K]\!]c = [\![K']\!]c$' helps to explore the similarities between the two special cases.

3.4 Population Refactoring

The PL refactoring notion discussed so far is useful to check safety when transforming a PL into another. We can then use it to justify the rightmost transformation in Fig. 2, whereas typical program refactorings justify the leftmost pair of transformations. The middle transformation, however, we cannot justify by either refactoring notion since we actually transform two programs, which we can see as two single-product PLs,[8] into a PL. To capture this merging situation, we introduce a refactoring notion that deals with more than one source PL. We have to guarantee that the target PL generates enough products to match the source PLs products. As before, we first define refinement. Here is the definition considering two source PLs.

Definition 8 ⟨Population refinement⟩
For product lines PL_1, PL_2, and PL, we say that the first two PLs are refined by the third, denoted by

$$PL_1\, PL_2 \sqsubseteq PL$$

whenever

$$PL_1 \sqsubseteq PL \wedge PL_2 \sqsubseteq PL$$

□

[8] See Sec. 2.4.

It is easy to generalize the definition for any number of source PLs, but we omit the details for simplicity. So, whereas we will likely use Definition 6 to indirectly relate product families (products with many commonalities and few differences), we will likely use Definition 8 to indirectly relate product populations (products with many commonalities but also with many differences) [vO02]. This assumes that, as a population has less commonality, it might have been initially structured as a number of PLs. In this case, the population corresponds to the union of the families generated by each source PL, or by the target PL.

Now we define population refactoring in a similar way to PL refactoring.

Definition 9 ⟨Population refactoring⟩
For product lines PL_1, PL_2, and PL, the first two are refactored by the third, denoted

$$PL_1\, PL_2 \lll PL$$

whenever

$$PL_1\, PL_2 \sqsubseteq PL$$

and PL is better than PL_1 and PL_2 with respect to quality factors such as reusability and maintainability.

□

Assuming proper FMs and CKs, population refinement justifies the middle transformation in Fig. 2. Refactoring follows from the fact that we eliminate duplicated code.

4 Product Line Refactoring Catalogue

With the refactoring notions introduced so far, we are able to derive a PL from existing products, and also to evolve a PL by simply improving its design or by adding new products while preserving existing ones. In this way we essentially handle the problems mentioned in Sec. 1. Nevertheless, it is useful to provide a catalog of common refactorings, so that developers do not need to reason directly about the definitions when evolving PLs. So in this section we illustrate different kinds of transformation templates[9] that are representative for deriving such a catalogue. We first introduce global transformations that affect a PL as a whole, changing FM, CK, and assets. Later we discuss transformations that affect PL elements in a separate and compositional way. In both cases we focus on refinement transformations, which are the essence of a refactoring catalogue, as refactoring transformations basically correspond to refinement transformations that consider quality improvement.

4.1 Overall Product Line Transformations

The first transformation we consider generalizes the refactoring illustrated by Fig. 7, where we add the optional Copy feature to an existing PL. Instead of

[9] Hereafter transformations.

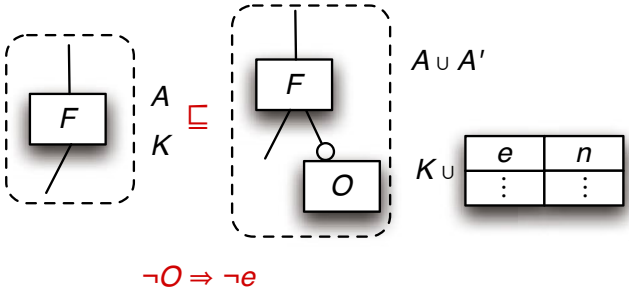


Fig. 9. Add optional feature refinement template

focusing on details of a specific situation like that, a transformation such as the one depicted by Fig. 9 precisely specifies, in an abstract way, that adding an optional feature is possible when the extra rows added to the original CK are only enabled by the selection of the new optional feature. We express this by the propositional logic precondition $\neg O \Rightarrow \neg e$, which basically requires e to be equivalent to propositions of the form O or $O \wedge e'$ for any feature expression e'. This assures that products built without the new feature correspond exactly to the original PL products. In this refinement transformation, we basically impose no constraints on the original PL elements; we only require the original FM to have at least one feature, identified in the transformation by the meta-variable F. We can extend the asset mapping as wished, provided that the result is a valid asset mapping. So in this case A' should not map names that are already mapped by A. Similarly, O should be a feature name that does not appear in the original FM, otherwise we would have an invalid FM in the target PL. For simplicity, we omit these constraints and always assume that the PLs involved in a transformation are valid.

When we need to refer to more details about the PL elements, we can use a more explicit notation, as illustrated in Fig. 10. For instance, n refers to the original set of names mapped to assets, and F now denotes the whole source FM. This establishes, in a more direct way, that it is not safe to change the original CK and asset mapping when adding an optional feature.

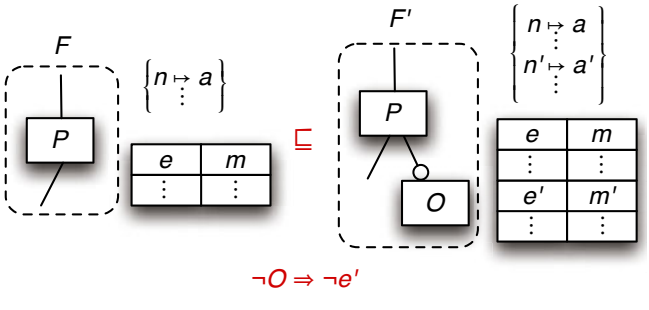


Fig. 10. Add optional feature detailed refinement template

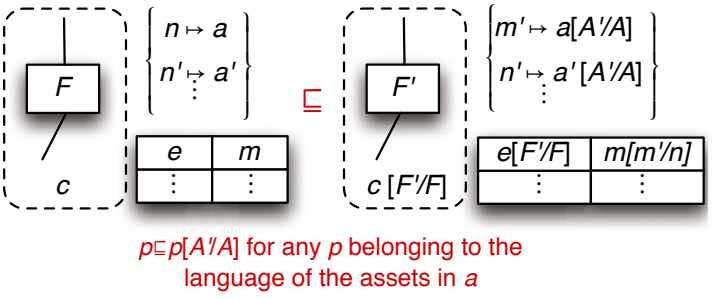


Fig. 11. PL renaming refinement template

We now discuss the PL renaming refinement transformation. Figure 11 generalizes the refactoring illustrated in Fig. 6, where we rename the Music feature and related artifacts to Audio. The transformation basically establishes that this kind of overall PL renaming is possible when renaming is a refinement in the involved core assets languages. Note that changing the name of F to F' requires changes to FM constraints ($c[F'/F]$) and CK feature expressions ($e[F'/F]$). Moreover, changing asset name n to m' requires changing the CK ($m[m'/n]$), whereas changing the asset declaration from A to A' implies changes to all declarations in a'.

Composing transformations. Besides elaborate transformations like PL renaming, which captures a major refactoring situation, it is useful to have simpler transformations that capture just one concern. In fact, from a comprehensive set of basic transformations we can, by composition, derive elaborate transformations. For example, the transformation in Fig. 12 focus on feature renaming whereas the one in Fig. 13 focus on renaming asset names, keeping the original feature names. Both do not have preconditions, so can always be applied provided the source pattern matches the source PL. By applying them in sequence and then applying asset declaration renaming (see Fig. 14), we derive the PL renaming transformation, which deals with all three concerns at once.

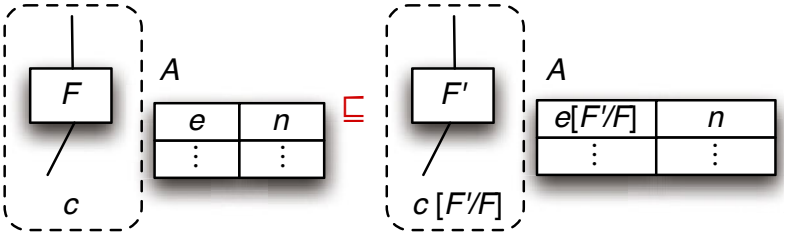


Fig. 12. Feature renaming refinement template

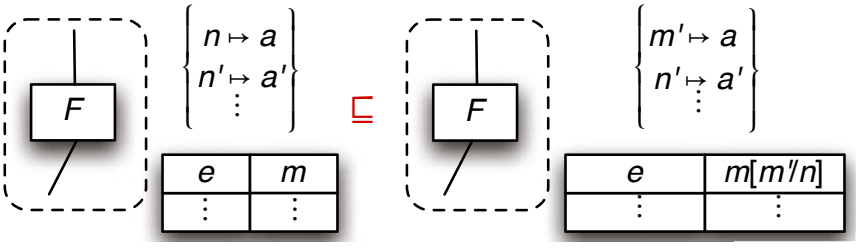


Fig. 13. Asset renaming refinement template

Whereas the elaborate transformations are useful for evolving product lines in practice, the basic transformations are useful for deriving a catalogue of elaborate transformations and verifying its soundness and completeness. It is, in fact, good practice to first propose basic transformations and then derive the elaborate ones, which are more appropriate for practical use [SB04, BSCC04].

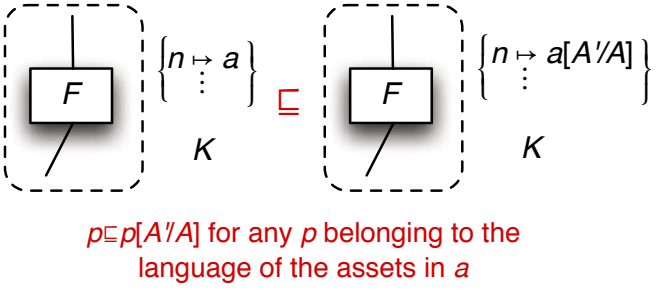


Fig. 14. Asset declaration renaming refinement template

By carefully looking at the transformation in Fig. 14, we notice that it focuses on evolving a single PL element: the asset mapping. In this case, the FM and CK are not impacted by the transformation. In general, when possible, it is useful to evolve PL elements independently, in a compositional way. So, besides the overall PL transformations illustrated so far, a refactoring catalogue should have transformations for separately dealing with FMs, CKs, and asset mappings, as illustrated in the following sections.

4.2 Asset Mapping Transformations

Transformations that focus on changing a single PL element (FM, CK, or asset mapping) should be based on specific refinements notions, not on the overall PL refinement notion. For asset mappings, exactly the same names should be mapped, not necessarily to the same assets, but to assets that refine the original ones.

Definition 10 ⟨Asset mapping refinement⟩
For asset mappings A and A', the first is refined by the second, denoted

$$A \sqsubseteq A'$$

whenever

$$\mathbf{dom}(A) = \mathbf{dom}(A') \wedge \forall a \in \mathbf{dom}(A) \cdot A(a) \sqsubseteq A'(a)$$

where $\mathbf{dom}(A)$ denotes the domain of A.

□

Note that $A(a) \sqsubseteq A'(a)$ in the definition refers to context independent asset refinement, not to program refinement.

Given this definition, we can propose transformations such as the one in Fig. 15, which renames assets declarations, provided that renaming is a refine-

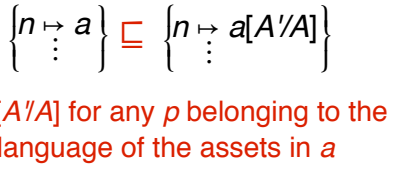


Fig. 15. Asset mapping refinement template

ment in the underlying asset languages. This essentially simplifies the transformation in Fig. 14, by focusing on changing only an asset mapping. We do not prove this here, but we can refine a PL by applying this kind of transformation and keeping FM and CK as in the source PL. This relies on the compositionality of asset refinement, as briefly discussed in Sec. 2.3, but we omit the details in order to concentrate on the different kinds of templates.

4.3 Feature Model Transformations

It is also useful to separately evolve feature models. We explore this in detail elsewhere [AGM+06, GMB08], but here we briefly illustrate the overall idea.[10] Instead of providing a refinement notion as in the previous section, we work with an equivalence. Two FMs are equivalent if they have the same semantics.

Definition 11 ⟨Feature model equivalence⟩
Feature models F and F' are equivalent, denoted $F \cong F'$, whenever $[\![F]\!] = [\![F']\!]$.

□

This equivalence is necessary for ensuring that separate modifications to a FM imply refinement for the PL as a whole. In fact, FM refinement requires only

[10] We use $\cong$ and $=$ to respectively denote semantic and syntactic equality of feature models, whereas in previous work we respectively use $=$ and $\equiv$.

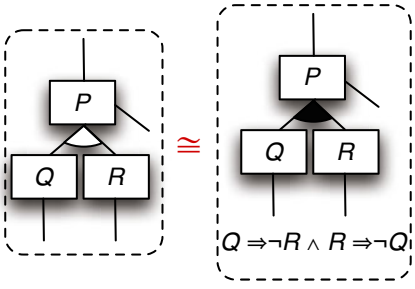


Fig. 16. Replace alternative equivalence template

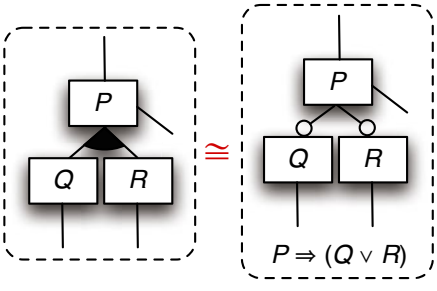


Fig. 17. Replace or equivalence template

$[\![F]\!] \subseteq [\![F']\!]$, but this allows the new FM to have extra configurations that might not generate valid programs; the associated FM refinement transformation would not lead to a valid PL. For example, consider that the extra configurations result from eliminating an alternative constraint between two features, so that they are now optional. The assets that implement these features might well be incompatible, generating an invalid program when we select both features. Refinement of the whole PL, in this case, would also demand changes to the assets and CK.

Such an equivalence allows us to propose pairs of transformations (one from left to right, and another from right to left) as in Fig. 16. From left to right, we have a transformation that replaces an *alternative* by an *or* relationship, with appropriate constraints. From right to left, we have a transformation that introduces an *alternative* relationship that was indirectly expressed by FM constraints.

Similarly, Fig. 17 shows how we express *or* features as *optional* features. From left to right we can notice a pattern of transforming a FM into constraints, whereas in the opposite direction we can see constraints expressed as FM graphical notation. This is further illustrated by the transformation in Fig. 18, which, from left to right, removes *optional* relationships. A comprehensive set of transformations like these allows us to formally derive other FM equivalences, which can be useful for simplifying FMs and, consequently, refactoring a PL.

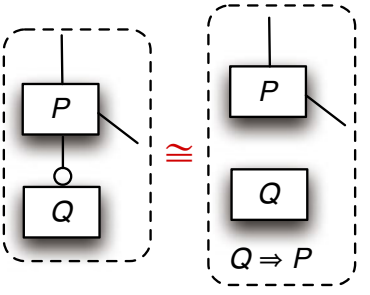


Fig. 18. Remove optional equivalence template

4.4 Configuration Knowledge Transformations

For configuration knowledge transformations, we also rely on an equivalence relation.

Definition 12 ⟨Configuration knowledge equivalence⟩
Configuration knowledge K is equivalent to K', denoted $K \cong K'$, whenever $[\![K]\!] = [\![K']\!]$. □

Due to the equivalence, we again have pairs of transformations. Figure 19 illustrates four such transformations, indicating that we can change rows in a CK (bottom left), apply propositional reasoning to feature expressions (top left), merge rows that refer to the same set of asset names (top right), and merge rows with equivalent expressions (bottom right). This is not a complete set of transformations, but gives an idea of what a comprehensive PL refactoring catalogue should contain.

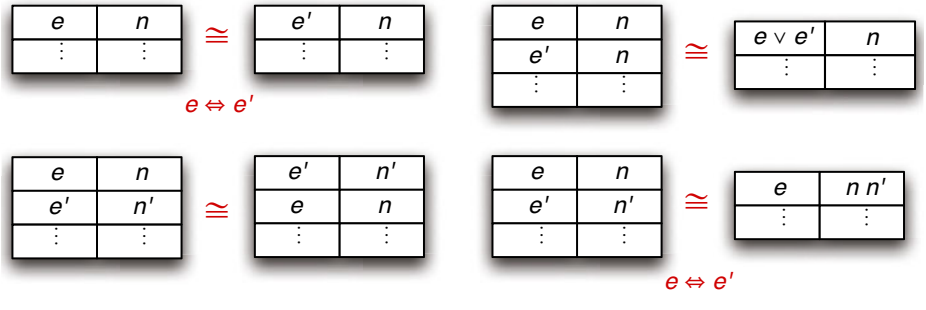


Fig. 19. Configuration knowledge equivalence templates

5 Conclusions

In this chapter we introduce and formalize notions of refinement and refactoring for software product lines. Based on these notions and specific ones for feature

models, configuration knowledge, and asset mappings, we also illustrate the kinds of transformation that should constitute a product line refactoring catalogue.

We hope this stimulates further work towards such a catalogue, considering different notations and semantic formalizations for software product line artifacts [CHE05, Bat05, SHTB07, GMB08]. This is important to guide and improve safety of the product line derivation and evolution processes. The presented transformation templates precisely specify the transformation mechanics and preconditions. This is especially useful for correctly implementing these transformations and avoiding typical problems with current program refactoring tools [ST09]. In fact, even subtler problems can appear with product line refactoring tools.

The ideas formalized here capture previous practical experience on product line refactoring [ACV+05, AJC+05, TBD06, KAB07], including development and use of a product line refactoring tool [CBS+07, ACN+08]. However, much still has to be done to better evaluate our approach and adapt existing processes and tools for product line refactoring.

Besides working towards a comprehensive refactoring catalogue, we hope to formally prove soundness and study completeness of product line transformations for the notations we use here for feature models and configuration knowledge. We should also discuss refinement and refactoring properties, which we omitted from this introduction to the subject.

Acknowledgements

I would like to thank the anonymous reviewers and colleagues of the Software Productivity Group for helping to significantly improve this work. Leopoldo Teixeira and Márcio Ribeiro provided interesting bad smells and refactoring scenarios from the Mobile Media example. Rodrigo Bonifácio played a fundamental role developing the configuration knowledge approach used here. Vander Alves, Rohit Gheyi, and Tiago Massoni carried on the initial ideas about feature model and product line refactoring. Fernando Castor, Carlos Pontual, Sérgio Soares, Rodrigo, Leopoldo, Rohit, and Márcio provided excellent feedback on early versions of the material presented here. Besides all, working with those guys is a lot of fun! I would also like to acknowledge current financial support from CNPq, FACEPE, and CAPES projects, and early support from FINEP and Meantime mobile creations, which developed the Rain of Fire product line.

References

[ACN+08] Alves, V., Calheiros, F., Nepomuceno, V., Menezes, A., Soares, S., Borba, P.: FLiP: Managing software product line extraction and reaction with aspects. In: 12th International Software Product Line Conference, p. 354. IEEE Computer Society, Los Alamitos (2008)

[ACV+05] Alves, V., Cardim, I., Vital, H., Sampaio, P., Damasceno, A., Borba, P., Ramalho, G.: Comparative analysis of porting strategies in J2ME games. In: 21st IEEE International Conference on Software Maintenance, pp. 123–132. IEEE Computer Society, Los Alamitos (2005)

[AGM+06] Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring product lines. In: 5th International Conference on Generative Programming and Component Engineering, pp. 201–210. ACM, New York (2006)

[AJC+05] Alves, V., Matos Jr., P., Cole, L., Borba, P., Ramalho, G.: Extracting and evolving mobile games product lines. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 70–81. Springer, Heidelberg (2005)

[Bat05] Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)

[BB09] Bonifácio, R., Borba, P.: Modeling scenario variability as crosscutting mechanisms. In: 8th International Conference on Aspect-Oriented Software Development, pp. 125–136. ACM, New York (2009)

[BSCC04] Borba, P., Sampaio, A., Cavalcanti, A., Cornélio, M.: Algebraic reasoning for object-oriented programming. Science of Computer Programming 52, 53–100 (2004)

[CB05] Cole, L., Borba, P.: Deriving refactorings for AspectJ. In: 4th International Conference on Aspect-Oriented Software Development, pp. 123–134. ACM, New York (2005)

[CBS+07] Calheiros, F., Borba, P., Soares, S., Nepomuceno, V., Alves, V.: Product line variability refactoring tool. In: 1st Workshop on Refactoring Tools, pp. 33–34 (July 2007)

[CE00] Czarnecki, K., Eisenecker, U.: Generative programming: methods, tools, and applications. Addison-Wesley, Reading (2000)

[CHE05] Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice 10(1), 7–29 (2005)

[CN01] Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading (2001)

[FCS+08] Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Filho, F., Dantas, F.: Evolving software product lines with aspects: an empirical study on design stability. In: 30th International Conference on Software Engineering, pp. 261–270. ACM, New York (2008)

[Fow99] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)

[GA01] Gacek, C., Anastasopoulos, M.: Implementing product line variabilities. SIGSOFT Software Engineering Notes 26(3), 109–117 (2001)

[GMB05] Gheyi, R., Massoni, T.M., Borba, P.: A rigorous approach for proving model refactorings. In: 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 372–375 (2005)

[GMB08] Gheyi, R., Massoni, T., Borba, P.: Algebraic laws for feature models. Journal of Universal Computer Science 14(21), 3573–3591 (2008)

[KAB07] Kastner, C., Apel, S., Batory, D.: A case study implementing features using AspectJ. In: 11th International Software Product Line Conference, pp. 223–232. IEEE Computer Society, Los Alamitos (2007)

[KCH+90] Kang, K., Cohen, S., Hess, J., Novak, W., Spencer Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)

[KHH+01] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: Getting started with AspectJ. Communications of the ACM 44(10), 59–65 (2001)

[Kru02] Krueger, C.: Easing the transition to software mass customization. In: van der Linden, F.J. (ed.) PFE 2002. LNCS, vol. 2290, pp. 282–293. Springer, Heidelberg (2002)

[MGB08] Massoni, T., Gheyi, R., Borba, P.: Formal model-driven program refactoring. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 362–376. Springer, Heidelberg (2008)

[Opd92] Opdyke, W.: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign (1992)

[PBvdL05] Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005)

[Rob99] Roberts, D.: Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana-Champaign (1999)

[SB04] Sampaio, A., Borba, P.: Transformation laws for sequential object-oriented programming. In: Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.) PSSE 2004. LNCS, vol. 3167, pp. 18–63. Springer, Heidelberg (2006)

[SHTB07] Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., Bontemps, Y.: Generic semantics of feature diagrams. Computer Networks 51(2), 456–479 (2007)

[ST09] Steimann, F., Thies, A.: From public to private to absent: refactoring Java programs under constrained accessibility. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 419–443. Springer, Heidelberg (2009)

[TBD06] Trujillo, S., Batory, D., Diaz, O.: Feature refactoring a multi-representation program into a product line. In: 5th International Conference on Generative Programming and Component Engineering, pp. 191–200. ACM, New York (2006)

[vdLSR07] van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering. Springer, Heidelberg (2007)

[vO02] van Ommering, R.C.: Building product populations with sofwtare components. In: 24th International Conference on Software Engineering, pp. 255–265. ACM, New York (2002)

Excerpts from the TXL Cookbook

James R. Cordy

School of Computing, Queen's University
Kingston ON, K7L 3N6, Canada
cordy@cs.queensu.ca
http://www.cs.queensu.ca/~cordy

Abstract. While source transformation systems and languages like DMS, Stratego, ASF + SDF, Rascal and TXL provide a general, powerful base from which to attack a wide range of analysis, transformation and migration problems in the hands of an expert, new users often find it difficult to see how these tools can be applied to their particular kind of problem. The difficulty is not that these very general systems are ill-suited to the solution of the problems, it is that the paradigms for solving them using combinations of the system's language features are not obvious.

In this paper we attempt to approach this difficulty for the TXL language in a non-traditional way - by introducing the paradigms of use for each kind of problem directly. Rather than simply introducing TXL's language features, we present them in context as they are required in the paradigms for solving particular classes of problems such as parsing, restructuring, optimization, static analysis and interpretation. In essence this paper presents the beginnings of a "TXL Cookbook" of recipes for effectively using TXL, and to some extent other similar tools, in a range of common source processing and analysis problems. We begin with a short introduction to TXL concepts, then dive right in to some specific problems in parsing, restructuring and static analysis.

Keywords: source transformation, source analysis, TXL, coding paradigms.

1 Introduction

Source transformation systems and languages like DMS [2], Stratego [6], ASF + SDF [3,5], Rascal [20] and TXL [8] provide a very general, powerful set of capabilities for addressing a wide range of software analysis and migration problems. However, almost all successful practical applications of these systems have involved the original authors or long-time experts with the tools. New potential users usually find it difficult and frustrating to discover how they can leverage these systems to attack the particular problems they are facing.

This is not an accident. These systems are intentionally designed to be be very general, and their features and facilities are therefore at a level of abstraction far from the level at which new users understand their particular problems. What

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 27–91, 2011.

they are interested in is not what the general language features are, but rather how they should be used to solve problems of the kind they are facing. The real core of the solution for any particular problem is not in the language or system itself, but rather in the paradigm for using it to solve that kind of problem.

In this paper we aim to address this issue head-on, by explicitly outlining the paradigms for solving a representative set of parsing, transformation and analysis problems using the TXL source transformation language. In the long run we are aiming at a "TXL Cookbook", a set of recipes for applying TXL to the many different kinds of problems for which it is well suited. While the paradigms are couched here in terms of TXL, in many cases the same paradigms can be used with other source transformation systems as well.

In what follows we begin with a short introduction to the basics of TXL, just to set the context, and then dive directly into some representative problems from four different problem domains: parsing, restructuring, optimization, and static and dynamic analysis. With each specific problem we outline the basic paradigms used in concrete solutions written in TXL. Along the way we discuss TXL's philosophy and implementation as they influence the solutions. Although it covers many issues, this set of problems is by no means complete, and it is expected that the cookbook will grow in future to be more comprehensive.

Our example problems are set in the context of a small, simple imperative language designed for the purpose of demonstrating transformation and analysis toolsets. The language, TIL ("Tiny Imperative Language") [11], was designed by Jim Cordy and Eelco Visser as the intended basis of a set of benchmarks for source transformation and analysis systems.

It is not our intention to cover the features of the TXL language itself here - there are already other published papers on the language [8] and programming in it [9], and features of the language are covered in detail in the TXL reference manual [10]. Rather, here we concentrate on the paradigms for solving problems using it, assuming only a basic knowledge.

2 TXL Basics

TXL [8] is a programming language explicitly designed for authoring source transformation tasks of all kinds. It has been used in a wide range of applications involving millions of lines of code [7]. Unlike most other source transformation tools, TXL is completely self-contained - all aspects of the source transformation, including the scanner, parser, transformer and output pretty-printer are all written in TXL. Because they have no dependencies on external parsers, frameworks or libraries, TXL programs are easily ported across platforms.

2.1 The TXL Paradigm

The TXL paradigm is the usual for source transformation systems (Figure 1). Input text is scanned and parsed into an internal parse tree, pattern-replacement rewriting rules are applied to the parse tree to transform it to another, and then the transformed tree is unparsed to the new output text.

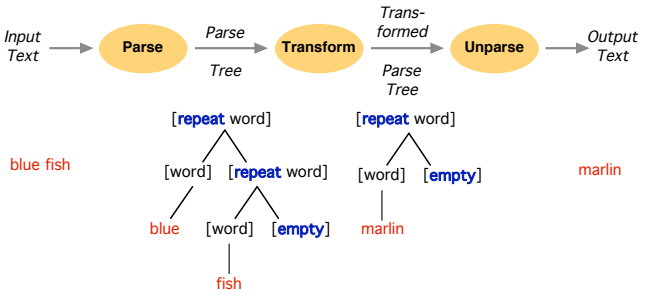


Fig. 1. The TXL Paradigm

Grammars and transformation rules are specified in the TXL language, which is efficiently implemented by the TXL Processor (Figure 2). The TXL processor is directly interpretive, with no parser generator or compile phase, which means there is little overhead and no maintenance barrier to running multiple transformations end-to-end. Thus TXL is well suited to test-driven development and rapid turnaround. But more importantly, transformations can be decomposed into independent steps with only a very small performance penalty, and thus most complex TXL transformations are architected as a sequence of successive approximations to the final result.

2.2 Anatomy of a TXL Program

A TXL program typically has three parts (Figure 3) : The *base grammar* defines the lexical forms (tokens) and the rooted set of syntactic forms (nonterminals or types) of the input language. Often the base grammar is kept in a separate file and included using an include statement. The *program* nonterminal is the root of the grammar, defining the form of the entire input. The optional *grammar overrides* extend or modify the syntactic forms of the grammar to allow output and intermediate forms of the transformation that are not part of the input language. Finally, the *rule set* defines the rooted set of transformation rules and functions to be applied to the input. The *main* rule or function is the root of the rule set, and is automatically applied to the entire input.

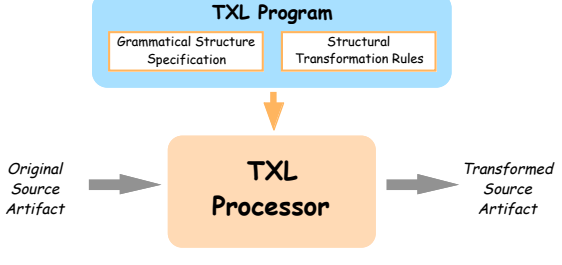


Fig. 2. The TXL Processor

Fig. 3. Parts of a TXL Program

While TXL programs are typically laid out as base grammar followed by overrides then rules, there is no ordering requirement and grammatical forms and rules can be mixed in the TXL program. To aid in readability, both grammars and rule sets are typically defined in topological order, starting from the *program* nonterminal and the *main* rule.

2.3 The Grammar: Specifying Lexical Forms

Lexical forms specify how the input text is partitioned into indivisible basic symbols (tokens or terminal symbols) of the input language. These form the basic types of the TXL program. The *tokens* statement gives regular expressions for each kind of token in the input language, for example, C hexadecimal integers:

```
tokens
    hexintegernumber    "0[xX][abcdefABCDEF\d]+"
end tokens
```

Tokens are referred to in the grammar using their name enclosed in square brackets (e.g., [hexintegernumber]). A set of default token forms are predefined in TXL, including C-style identifiers [id], integer and floating point numbers [number], string literals [stringlit], and character literals [charlit].

The *comments* statement specifies the commenting conventions of the input language, that is, sections of input source that are to be considered as commentary. For example, C commenting conventions can be defined as follows:

```
comments
    /*   */
    //
end comments
```

By default, comments are ignored (treated as white space) by TXL, but they can be treated as significant input tokens if desired. Most analysis tasks can ignore comments, but transformation tasks may want to preserve them.

The *keys* statement specifies that certain identifiers are to be treated as unique special symbols (and not as identifiers), that is, keywords of the input language. For example, the following could be used to specify the keywords of a subset of standard Pascal. The "end" keyword must be quoted (preceded by a single quote) to distinguish it from TXL's own end keyword. In general, TXL's own keywords and special symbols are the only things that need to be quoted in TXL, and other words and symbols simply represent themselves.

```
keys
    program procedure function
    repeat until for while do begin 'end
end keys
```

The *compounds* statement specifies character sequences to be treated as a single character, that is, compound tokens. Since "%" is TXL's end-of-line comment character, symbols containing percent signs must be quoted in TXL programs. Compounds are really just a shorthand for (unnamed) token definitions.

```
compounds
    :=  <=  >=  ->  <->  '%=          % note quoted "%"
end compounds
```

2.4 The Grammar: Specifying Syntactic Forms

Syntactic forms (nonterminal symbols or *types*) specify how sequences of input tokens are grouped into the structures of the input language. These form the structured types of the TXL program - In essence, each TXL program defines its own symbols and type system. Syntactic forms are specified using an (almost) unrestricted ambiguous context free grammar in extended BNF notation, where:

```
 X      literal terminal symbols (tokens) represent themselves
[X]     terminal (token) and nonterminal types appear in brackets
 |      or bar separates alternative syntactic forms
```

Each syntactic form is specified using a *define* statement. The special type [program] describes the structure of the entire input. For example, here is a simple precedence grammar for numeric expressions:

```
File "Expr.grm"

    define program    % goal symbol of input
        [expression]
    end define

    define expression
          [term]
        | [expression] + [term]
        | [expression] - [term]
    end define

    define term
          [primary]
        | [term] * [primary]
        | [term] / [primary]
    end define

    define primary
          [number]
        | ( [expression] )
    end define
```

Grammars are most efficient and natural in TXL when most user-oriented, using sequences in preference to recursion, and simpler forms rather than semantically distinguished cases. In general, *yacc*-style compiler "implementation" grammars should be avoided.

Sequences and optional items can be specified using an extended BNF-like sequence notation:

```
[repeat X]  or  [X*]        sequence of zero or more (X*)
[repeat X+] or  [X+]        sequence of one or more (X+)
[list X]    or  [X,]        comma-separated list of zero or more
[list X+]   or  [X,+]       comma-separated list one or more
[opt X]     or  [X?]        optional (zero or one)
```

For more natural patterns in transformation rules, these forms should always be used in preference to hand-coded recursion for specifying sequences in grammars, since TXL is optimized for handling them.

Formatting cues in defines specify how output text is to be formatted:

```
[NL]    newline  in unparsed output
[IN]    indent subsequent unparsed output by four spaces
[EX]    exdent subsequent unparsed output by four spaces
```

Formatting cues have no effect on input parsing and transformation patterns.

2.5 Input Parsing

Input is automatically tokenized and parsed according to the grammar. The entire input must be recognizable as the type [program], and the result is represented internally as a parse tree representing the structural understanding of the

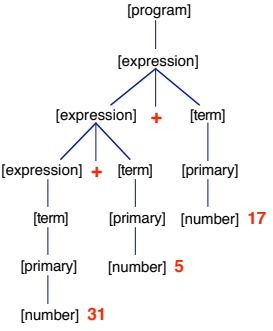


Fig. 4. Parse tree for the expression 31+5+17 according to the example grammar

input according to the grammar. Figure 4 shows the parse tree resulting from the input of the numeric expression "31+5+17" to a TXL program using the TXL grammar shown above.

All pattern matching and transformation operations in TXL rules and functions work on the parse tree. Since each TXL program defines its own grammar, it is important to remember that syntax errors in the input may indicate an incorrect grammar rather than a malformed input.

2.6 Base Grammars and Overrides

The base grammar for the syntax of the input language is normally kept in a separate grammar file which is rarely if ever changed, and is included in the TXL program using a TXL *include* statement. While the base grammar itself can serve for many purposes, since TXL is strongly typed in the type system of the grammar (that is, all trees manipulated or produced by a TXL transformation rule must be valid instances of their grammatical type, so that malformed results cannot be created), it is often necessary to add output or intermediate forms. Analysis and transformation tasks that require output or intermediate forms not explained by the grammar can add their own forms using *grammar overrides*.

Grammar overrides modify or extend the base grammar's lexical and syntactic forms by modifying existing forms using *redefine* statements. Redefine statements can completely replace an original form, for example, this redefine of [primary] in the example grammar above will modify it to allow identifiers and lists of expressions:

```
include "Expr.grm"          % the original example grammar

redefine primary
    [id]
  | [number]
  | ( [expression,+] )
end redefine
```

The semantics of such a redefine is that the original form is replaced by the new form in the grammar.

Grammar overrides can also extend the existing forms of a grammatical type, using the notation "..." to refer to the original definition of the nonterminal type. For example, this redefine will allow XML markup on any [statement] by extending the definition of [statement] to allow a marked-up statement.

```
include "C.grm"                    % the C language grammar

redefine statement
    ...                            % includes the original forms
  | <[id]> [statement] </[id]>     % adds the new XML markup form
end redefine
```

The redefine says that a statement can be any form it was before ("..."), or the new form. "..." is not an elision here, it is part of the TXL language syntax, meaning "whatever [statement] was before".

2.7 Transformation Rules

Once the input is parsed, the actual input to output source transformation is specified in TXL using a rooted set of transformation *rules*. Each transformation rule specifies a *target type* to be transformed, a *pattern* (an example of the particular instance of the type that we are interested in replacing) and a *replacement* (an example of the result we want when we find such an instance).

```
% replace every 1+1 expression with 2
rule addOnePlusOne
    replace [expression]      % target type to search for
        1 + 1                 % pattern to match
    by
        2                     % replacement to make
end rule
```

TXL rules are strongly typed - that is, the replacement must be of the same grammatical type as the pattern (that is, the target type). While this seems to preclude cross-language and cross-form transformations, as we shall see, because of grammar overrides this is not the case!

The pattern can be thought of as an actual source text example of the instances we want to replace, and when programming TXL one should think by example, not by parse tree. Patterns consist of a combination of tokens (input symbols, which represent themselves) and named variables (tagged nonterminal types, which match any instance of the type). For example, the TXL variable N1 in the pattern of the following rule will match any item of type [number] :

```
rule optimizeAddZero
    replace [expression]
        N1 [number] + 0
    by
        N1
end rule
```

When the pattern is matched, variable names are bound to the corresponding item of their type in the matched instance of the target type. Variables can be used in the replacement to copy their bound instance into the result, for example the item bound to N1 will be copied to the replacement of each [expression] matching the pattern of the rule above.

Similarly, the replacement can be thought of as a source text example of the desired result. Replacements consist of tokens (input symbols, which represent themselves) and references to bound variables (using the tag name of the variable from the pattern). References to bound variables in the replacement denote copying of the variable's bound instance into the result.

References to variables can be optionally further transformed by subrules (other transformation rules), which further transform (only) the copy of the variable's bound instance before it is copied into the result. Subrules are applied to a variable reference using postfix square bracket notation X[f], which in functional notation would be f(X). X[f][g] denotes functional composition of subrules - that is, g(f(X)). For example, this rule looks for instances of expressions (including subexpressions) consisting of a number plus a number, and resolves the addition by transforming copy of the first number using the [+] subrule to add the second number to it. ([+] is one of a large set of TXL built-in functions.)

```
rule resolveAdditions
    replace [expression]
        N1 [number] + N2 [number]
    by
        N1 [+ N2]
end rule
```

When a rule is applied to a variable, we say that the variable's copied value is the rule's *scope*. A rule application only transforms inside the scope it is applied to. The distinguished rule called *main* is automatically applied to the entire input as its scope - any other rules must be explicitly applied as subrules to have any effect. Often the main rule is a simple *function* to apply other rules:

```
function main
    replace [program]
        EntireInput [program]
    by
        EntireInput [resolveAdditions] [resolveSubtractions]
                    [resolveMultiplys] [resolveDivisions]
end function
```

2.8 Rules and Functions

TXL has two kinds of transformation rules, *rules* and *functions*, which are distinguished by whether they should transform only one (for functions) or many (for rules) occurrences of their pattern. By default, rules repeatedly search their scope for the first instance of their target type matching their pattern, transform it to yield a new scope, and then reapply to the entire new scope until no more matches are found. By default, functions do not search, but attempt to match only their entire scope to their pattern, transforming it if it matches. For example, this function will match only if the entire expression it is applied to is a number plus a number, and will not search for matching subexpressions:

```
function resolveEntireAdditionExpression
    replace [expression]
        N1 [number] + N2 [number]
    by
        N1 [+ N2]
end function
```

Searching functions, denoted by "replace *", search to find and transform the first occurrence of their pattern in their scope, but do not repeat. Searching functions are used when only one match is expected, or only the first match should be transformed.

```
function resolveFirstAdditionExpression
    replace * [expression]
        N1 [number] + N2 [number]
    by
        N1 [+ N2]
end function
```

2.9 Rule Parameters

Rules and functions may be passed parameters, which bind the values of variables in the applying rule to the formal parameters of the subrule. Parameters can be used to build transformed results out of many parts, or to pass global context into a transformation rule or function. In this example, the [resolveConstants] outer rule finds a Pascal named constant declaration, and passes both the name and the value to the subrule [replaceByValue] which replaces all references to the constant name in the following statements by its value. The constant declaration is then removed by [resolveConstants] in its replacement.

```
rule resolveConstants
    replace [statement*]
        const C [id] = V [primary];
        RestOfScope [statement*]
    by
        RestOfScope [replaceByValue C V]
end rule

rule replaceByValue ConstName [id] Value [primary]
    replace [primary]
        ConstName
    by
        Value
end rule
```

2.10 Patterns and Replacements

The example-like patterns and replacements in rules and functions are parsed using the grammar in the same way as the input, to make pattern-tree / replacement-tree pairs. Figure 5 shows an example of the pattern and replacement trees for the [resolveAdditions] example rule. While sometimes it is helpful to be aware of the tree representation of patterns, in general it is best to think at the source level in a by-example style when programming TXL.

Rules are implemented by searching the scope parse tree for tree pattern matches of the pattern tree, and replacing each matched instance with a corresponding instantiation of the replacement tree. In Figure 6 we can see the sequence of matches that the rule [resolveAdditions] will find in the parse tree for the input expression "36+5+17". It's important to note that the second match does not even exist in the original scope - it only comes to be after the first replacement. This underlines the semantics of TXL rules, which search for

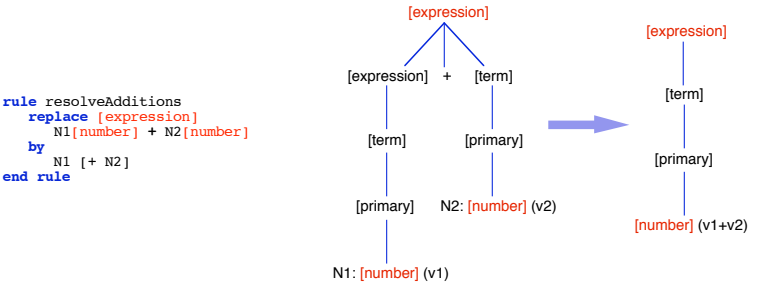


Fig. 5. Pattern and replacement trees for the [resolveAdditions] rule

one match at a time in their scope, and following a replacement, search the entire new scope for the next match.

Patterns may refer to a previously bound variable later in the same pattern (technically called *strong pattern matching*). This parameterizes the pattern with a copy of the bound variable, to specify that two parts of the matching instance must be the same in order to have a match. For example, the following rule's pattern matches only expressions consisting of the addition of two identical subexpressions (e.g., 1+1, 2*4+2*4, and (3-2*7)+(3-2*7)).

```
rule optimizeDoubles
    replace [expression]
        E [term] + E
    by
        2 * E
end rule
```

Patterns can also be parameterized by formal parameters of the rule, or other bound variables, to specify that matching instances must contain an identical copy of the variable's bound value at that point in the pattern. (We saw an example in the [replaceByValue] rule on the previous page.) A simple way to think about TXL variables is that references to a variable always mean a copy of its bound value, no matter what the context is.

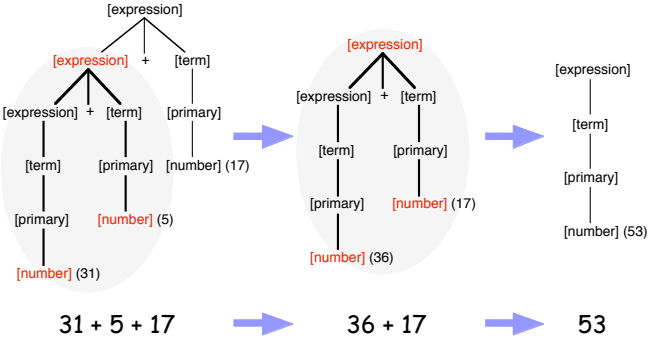


Fig. 6. Example application of the [resolveAdditions] rule

2.11 Deconstructors and Constructors

Patterns can be piecewise refined to more specific patterns using *deconstruct* clauses. Deconstructors specify that the deconstructed variable's bound value must match the given pattern - if not, the entire pattern match fails. Deconstructors act like functions - by default, the entire bound value must match the deconstructor's pattern, but "deconstruct *" (a *deep* deconstruct) searches within the bound value for a match. The following example demonstrates both kinds - a deep deconstruct matches the [if_condition] in the matched IfStatement, and the second deconstruct matches the entire IfCond only if it is exactly the word *false*.

```
rule optimizeFalseIfs
    replace [statement*]
        IfStatement [if_statement] ;
        RestOfStatements [statement*]
    deconstruct * [if_condition] IfStatement
        IfCond [if_condition]
    deconstruct IfCond
        'false
    by
        RestOfStatements
end rule
```

Pattern matches can also be constrained using *where* clauses, which allow for arbitrary matching conditions to be tested by subrules. The where clause succeeds only if its subrules find a match of their patterns. Like deconstructors, if a where clause fails, the entire pattern match fails. Here's an example use of where clauses to test that two sequential assignment statements do not interfere with each other (and thus the pair can be parallelized):

```
rule vectorizeScalarAssignments
    replace [statement*]
        V1 [variable] := E1 [expression];
        V2 [variable] := E2 [expression];
        RestOfScope [statement*]
    where not
        E2 [references V1]
    where not
        E1 [references V2]
    by
        < V1,V2 > := < E1,E2 > ;
        RestOfScope
end rule
```

While where clauses are more general, for efficiency reasons it is always better to use a deconstruct than a where clause when possible. Where clauses use a special kind of rule called a *condition rule*, for example [references] in the example above.

```
function references V [variable]
    deconstruct * [id] V
        Vid [id]
    match * [id]
        Vid
end function
```

Condition rules are different in that they have only a (possibly very complex) pattern, but no replacement - they simply succeed or fail to match their

pattern, but do not make any replacement. In this case, [references] looks inside its expression scope to see if there are any uses of the main identifier naming the variable it is passed.

Replacements can also be piecewise refined, to construct results from several independent pieces using *construct* clauses. Constructors allow partial results to be bound to new variables, allowing subrules to further transform them in the replacement or other constructors. In the example below, NewUnsortedSequence is constructed so that it can be further transformed by the subrule [sortFirstIntoPlace] in the replacement.

```
rule addToSortedSequence NewNum [number]
    replace [number*]
        OldSortedSequence [number*]
    construct NewUnsortedSequence [number*]
        NewNum OldSortedSequence
    by
        NewUnsortedSequence [sortFirstIntoPlace]
end rule
```

Even when constructors are not really required, constructing a complex replacement in well-named pieces can aid in readability of the rule.

This ends our basic introduction to TXL. We now move on to the real focus of this paper - the paradigms for solving real parsing, analysis and transformation problems using it. We begin by introducing TIL, the toy example language used as a platform for our example problems.

3 The TIL Chairmarks

TIL (Tiny Imperative Language) is a very small imperative language with assignments, conditionals, and loops, designed by Eelco Visser and James Cordy as a basis for small illustrative example transformations. All of the example applications in the TXL Cookbook work on TIL or extended dialects of it. Figure 7 shows two examples of basic TIL programs.

The TIL Chairmarks [12] are a small set of benchmark transformation and analysis tasks based on TIL. They are called “chairmarks” because they are too

File "factors.til"

```
// Find factors of a given number
var n;
write "Input n please";
read n;
write "The factors of n are";
var f;
f := 2;
while n != 1 do
    while (n / f) * f = n do
        write f;
        n := n / f;
    end;
    f := f + 1;
end;
```

File "multiples.til"

```
// First 10 multiples of numbers 1 through 9
for i := 1 to 9 do
for j := 1 to 10 do
    write i*j;
end; end;
```

Fig. 7. Example TIL programs

small to be called "benchmarks". These tasks form the basis of our cookbook, and the examples in this tutorial are TXL solutions to some of the problems posed in the Chairmarks. The TIL Chairmark problems are split into six categories: parsing, restructuring, optimization, static and dynamic analysis, and language implementation. In this tutorial we only have room for one or two specific problems from each category. In each case, a specific concrete problem is proposed and a TXL solution is demonstrated, introducing the corresponding TXL solution paradigms and additional language features as we go along. We begin with the most important category: parsing.

4 Parsing Problems

Every transformation or analysis task begins with the creation or selection of a TXL grammar for the source language. The form of the language grammar has a big influence on the ease of writing transformation rules. In these first problems, we explore the creation of language grammars, pretty-printers and syntactic extensions using the parsing aspect of TXL only, with no transformations. The grammars we create here will serve as the basis of the transformation and analysis problems in the following sections. In many cases, a TXL grammar is already available for the language you want to process on the TXL website.

It is important to remember that the purpose of the TXL grammar for an input language is not, in general, to serve as a syntax checker (unless of course that is what we are implementing). We can normally assume that inputs to be transformed are well-formed. This allows us to craft grammars that are simpler and more abstract than the true language grammar, for example allowing all statements uniformly everywhere in the language even if some are semantically valid only in certain contexts, such as the *return* statement in Pascal, which is valid only inside procedures. In general, such uniformity in the grammar makes analyzing and transforming the language forms easier. In the case of TIL, the language is simple enough that such simplification of the grammar is unnecessary.

4.1 Basic Parser / Syntax Checker

In this first problem, our purpose is only to create a grammar for the language we plan to process, in this case TIL. Figure 8 shows a basic TXL grammar (file "TIL.grm") for TIL. The main nonterminal of a TXL grammar must always be called [program], and there must be a nonterminal definition for [program] somewhere in the grammar. Implementing a parser and syntax checker using this grammar is straightforward, simply including the grammar in a TXL program that does nothing but match its input (Figure 9, file "TILparser.txl").

Paradigm. *The grammar is the parser.* TXL "grammars" are in some sense misnamed - they are not really grammars in the usual BNF specification sense, to be processed and analyzed by a parser generator such as SDF or Bison. Rather, a TXL grammar is a directly interpreted recursive descent parser, written in grammatical style. Thus in TXL the grammar is really a program for parsing

```
File "TIL.grm"

% TXL grammar for Tiny Imperative Language

% When pretty-printing, we parse and output
% comments, controlled by this pragma
% #pragma -comment

% Keywords of TIL, a reserved-word language
keys
    var if then else while do for
    read write 'end
end keys

% Compound tokens to be recognized
% as a single lexical unit
compounds
    :=  !=  <=  >=
end compounds

% Commenting convention for TIL -
% comments are ignored unless -comment is set
comments
    //
end comments

% Direct TXL encoding of the TIL grammar.
% [NL], [IN] and [EX] on the right are
% optional pretty-printing cues
define program
    [statement*]
end define

define statement
        [declaration]
    |   [assignment_statement]
    |   [if_statement]
    |   [while_statement]
    |   [for_statement]
    |   [read_statement]
    |   [write_statement]
    |   [comment_statement]
end define

% Untyped variables
define declaration
    'var [name] ;                [NL]
end define

define assignment_statement
    [name] := [expression] ;     [NL]
end define

define if_statement
    'if [expression] 'then       [IN][NL]
        [statement*]             [EX]
    [opt else_statement]
    'end ';                      [NL]
end define

define else_statement
    'else                        [IN][NL]
        [statement*]             [EX]
end define

define while_statement
    'while [expression] 'do      [IN][NL]
        [statement*]             [EX]
    'end ';                      [NL]
end define

define for_statement
    'for [name] := [expression]
            'to [expression] 'do       [IN][NL]
        [statement*]                   [EX]
    'end ';                            [NL]
end define

define read_statement
    'read [name] ;                     [NL]
end define

define write_statement
    'write [expression] ;              [NL]
end define

define comment_statement
    % Only ever present if -comment is set
    [NL] [comment] [NL]
end define

% Traditional priority expression grammar
define expression
        [comparison]
    |   [expression] [logop] [comparison]
end define

define logop
    'and | 'or
end define

define comparison
        [term]
    |   [comparison] [eqop] [term]
end define

define eqop
    = | != | > | < | >= | <=
end define

define term
        [factor]
    |   [term] [addop] [factor]
end define

define addop
    + | -
end define

define factor
        [primary]
    |   [factor] [mulop] [primary]
end define

define mulop
    * | /
end define

define primary
        [name]
    |   [literal]
    |   ( [expression] )
end define

define literal
        [integernumber]
    |   [stringlit]
end define

define name
    [id]
end define
```

Fig. 8. TIL grammar in TXL

File "TILparser.txl"

```
% TXL parser for Tiny Imperative Language

% All TXL parsers are automatically also pretty-printers if the
% grammar includes the optional formatting cues, as in this case

% Use the TIL grammar
include "TIL.grm"

% No need to do anything except recognize the input, since the grammar
% includes the output formatting cues

function main
    match [program]
        _ [program]
end function
```

Fig. 9. TIL parser and pretty-printer

the input language, where the input is source text and the output is a parse tree. When crafting TXL grammars, one needs to be aware of this fact, and think (at least partly) like a programmer rather than a language specifier.

The creation of a TXL grammar begins with the specification of the lexical forms (tokens, or terminal symbols) of the language, using TXL's regular expression pattern notation. Several common lexical forms are built in to TXL, notably [id], which matches C-style identifiers, [number], which matches C-style integer and float constants, [stringlit], which matches double-quoted C-style string literals, and [charlit], which matches single-quoted C-style character literals.

The TIL grammar uses only the default tokens [id], [integernumber] and [stringlit] as its terminal symbols, thus avoiding defining any token patterns of its own. ([integernumber] is a built-in refinement of [number] to non-floating point forms.) More commonly, it would be necessary to define at least some of the lexical forms of the input language explicitly using TXL *tokens* statements.

The TIL keywords are specified in the grammar using the *keys* statement, which tells TXL that the given words are reserved and not to be mistaken for identifiers. The *compounds* section tells us that the TIL symbols := and != are to be treated as single tokens, and the *comments* section tells TXL that TIL comments begin with // and go to the end of line. Comments are by default ignored and removed from the parsed input, and do not appear in the parse tree or output. However, they can be preserved (see section 4.2).

Paradigm. *Use sequences, not recursions.* The fact that TXL grammars are actually parsing programs has a strong influence on the expression of language forms. For example, in general it is better to express sequences of statements or expressions directly as sequences ([X*] or equivalently [repeat X]) rather than right- or left-recursive productions. This is both because the parser will be more efficient, and because the TXL pattern matching engine is optimized for searching sequences. Thus forms such as this one, which is often seen in BNF grammars, should be converted to sequences (in this case [statement*]) in TXL:

```
statements -> statement
           |  statements statement
```

Paradigm. *Join similar forms.* In order to avoid backtracking, multiple similar forms are typically joined together into one in TXL grammars. For example, when expressed in traditional BNF, the TIL grammar shows two forms for the if statement, with and without an else clause, as separate cases.

```
if_statement -> "if" expression "then"
                    statement*
                "end" ";"
              | "if" expression "then"
                    statement*
                "else"
                    statement*
                "end" ";"
```

While we could have coded this directly into the TXL grammar, because it is directly interpreted, when instances of the second form were parsed, TXL would have to begin parsing the first form until it failed, then backtrack and start over trying the second form to match the input. When many such similar forms are in the grammar, this backtracking can become expensive, and it is better to avoid it by programming the grammar more efficiently in TXL. In this case, both forms are subsumed into one by separating and making the else clause optional in the TXL define for [if_statement] (Figure 8).

Paradigm. *Encode precedence and associativity directly in the grammar.* As in all direct top-down parsing methods, left-recursive nonterminal forms can be a particular problem and should in general be avoided. However, sometimes, as with left-associative operators, direct left-recursion is required, and TXL will recognize and optimize such direct left-recursions. An example of this can be seen in the expression grammar for TIL (Figure 8), which encodes precedence and associativity of operators directly in the grammar using a traditional precedence chain. Rather than separate precedence and associativity into separate disambiguation rules, TXL normally includes them in the grammar in this way.

Figure 9 shows a TXL program using the TIL grammar that simply parses input programs, and the result of running it on the example program "multiples.til" of Figure 7, using the command:

```
txl -xml multiples.til TILparser.txl
```

is shown in Figure 10. The "-xml" output shows the internal XML form of the parse tree of the input program.

4.2 Pretty Printing

The next problem we tackle is creating a pretty-printer for the input language, in this case TIL. Pretty-printing is a natural application of source transformation systems since they all have to create source output of some kind.

Paradigm. *Using formatting cues to control output format.* TXL is designed for pretty-printing, and output formatting is controlled by inserting formatting cues for indent [IN], exdent [EX] and new line [NL] into the grammar. These cues look like nonterminal symbols, but have no effect on input parsing. Their

```
linux% txl multiples.til TILparser.txl -xml

<program>
 <repeat statement>
  <statement><for_statement> for
    <name><id>i</id></name> :=
    <expression><primary><literal><integernumber>1</integernumber></literal></primary></expression> to
    <expression><primary><literal><integernumber>9</integernumber></literal></primary></expression> do
    <repeat statement>
     <statement><for_statement> for
       <name><id>j</id></name> :=
       <expression><primary><literal><integernumber>1</integernumber></literal></primary></expression> to
       <expression><primary><literal><integernumber>10</integernumber></literal></primary></expression> do
       <repeat statement>
        <statement><write_statement> write
          <expression>
           <expression><primary><name><id>i</id></name></primary></expression>
           <op>*</op>
           <expression><primary><name><id>j</id></name></primary></expression>
          </expression> ;
         </write_statement>
        </statement>
       </repeat statement> end ;
      </for_statement>
     </statement>
    </repeat statement> end ;
   </for_statement>
  </statement>
 </repeat statement>
</program>
```

Fig. 10. Example XML parse tree output of TIL parser

only role is to specify how output is to be formatted. For example, in the TIL grammar of Figure 8, the definition for [while_statement] uses [IN][NL] following the while clause, specifying that subsequent lines should be indented, and that a new line should begin following the clause. The [EX] after the statements in the body specifies that subsequent lines should no longer be indented, and the [NL] following the end of the loop specifies that a new line should begin following the while statement.

Paradigm. *Preserving comments in output.* By default TXL ignores comments specified using the *comments* section as shown in Figure 8, where TIL comments are specified as from // to end of line. In order to preserve comments in output, we must tell TXL that we wish to do that using the *-comment* command-line argument or the equivalent #pragma directive,

```
#pragma -comment
```

Once we have done that, comments become first-class tokens and the grammar must allow comments anywhere they may appear. For well-formed input code this is not difficult, but in general it is tricky and can require significant tuning. It is a weakness of TXL that it has no other good way to preserve comments.

In the TIL case, we have only end-of-line comments and we will assume that they are used only at the statement level - if we observe other cases, they can be added to the grammar. This is specified in the grammar with the statement

form [comment_statement], (which has no effect when *-comment* is off because no [comment] token will be available to parse). [comment_statement] is defined to put a new line before each comment, in order to separate it in the output:

```
define comment_statement
    [NL] [comment] [NL]
end define
```

Figure 11 shows the result of pretty-printing *multiples.til* using the parsing program of Figure 9.

4.3 Language Extensions

Language extensions, dialects and embedded DSLs are a common application of source transformation systems. The next problem involves implementing a number of syntactic extensions to the TIL grammar. Syntactic extension is one of the things TXL was explicitly designed for, and the paradigm is straightforward.

Figure 12 shows four small language extensions to TIL, the addition of begin-end statements, the addition of arrays, the addition of functions, and the addition of modules (i.e., anonymous or singleton classes). New grammatical forms, tokens and keywords are defined using the usual tokens, keys and define statements of TXL, as for example with the [begin_statement] definition in the begin-end extension of TIL and the addition of the *function* keyword in the function extension of TIL (both in Figure 12).

Paradigm. *Extension of grammatical forms.* New forms are integrated into the existing language grammar using redefinitions of existing forms, such as [statement] in the begin-end dialect of TIL. TXL's *redefine* statement is explicitly designed to support language modifications and extensions. In the begin-end extension we can see the use of redefine to add a new statement form to an existing language:

```
redefine statement
        ...                     % refers to all existing forms
    |   [begin_statement]       % add alternative for our new form
end redefine
```

```
linux%  cat Examples/multiples.til

    // Output first 10 multiples of numbers 1 through 9
    for i:=1 to 9 do for j:=1 to 10 do
        // Output each multiple
    write i*j; end; end;

linux%  txl -comment multiples.til TILparser.txl

    // Output first 10 multiples of numbers 1 through 9
    for i := 1 to 9 do
        for j := 1 to 10 do
            // Output each multiple
            write i * j;
        end;
    end;
```

Fig. 11. Example output of the TIL pretty-printer

File "TILbeginend.grm"

```
% TXL grammar overrides for begin-end
% extension of the Tiny Imperative Language

% Add begin-end statements
redefine statement
        ...                 % existing forms
    |   [begin_statement]   % adds new form
end redefine

define begin_statement
    'begin                  [IN][NL]
        [statement*]        [EX]
    'end                    [NL]
end define
```

File "TILfunctions.grm"

```
% TXL grammar overrides for functions
% extension of the Tiny Imperative Language

% Add functions using grammar overrides

redefine declaration
        ...                     % existing
    |   [function_definition]   % new form
end redefine

redefine statement
        ...                     % existing
    |   [call_statement]
end redefine

keys
    'function
end keys

define function_definition
    'function [name] '( [name,] ')
                [opt colon_id]      [IN][NL]
        [statement*]                [EX]
    'end;                           [NL][NL]
end define

define call_statement
  [opt id_assign]
      [name] '( [expression,] ') ';  [NL]
end define

define colon_id
        ': [name]
end define

define id_assign
        [name] ':=
end define
```

File "TILarrays.grm"

```
% TXL grammar overrides for array
% extension of the Tiny Imperative Language

% Add arrays using grammar overrides
redefine declaration
      'var [name] [opt subscript] '; [NL]
   |  ...
end redefine

redefine primary
      [name] [opt subscript]
   |  ...
end redefine

redefine assignment_statement
    [name] [opt subscript] ':=
                    [expression] ';  [NL]
end redefine

define subscript
    '[ [expression] ']
end define
```

File "TILmodules.grm"

```
% TXL grammar overrides for module
% extension of the Tiny Imperative Language

% Add modules using grammar overrides
% Requires functions extension

redefine declaration
        ...                     % existing forms
    |   [module_definition]     % add new form
end redefine

keys
    'module  'public
end keys

define module_definition
    'module [name]              [IN] [NL]
        [statement*]            [EX]
    'end ;                      [NL] [NL]
end define

redefine function_definition
    [opt 'public] ...
end redefine
```

Fig. 12. TXL overrides for four dialects of TIL

Such a grammar modification is called a *grammar override* in TXL since it "overrides" or replaces the original definition with the new one. The "..." notation in this example is not an elision, it is an actual part of the TXL syntax. It refers to all of the previously defined alternative forms for the nonterminal type, in this case [statement], and is a shorthand for repeating them all in the redefine. It also makes the redefinition largely independent of the base grammar, so if the definition of [statement] in TIL changes, the language extension will not require any change, it will just inherit the new definition.

Because TXL grammars are actually directly interpreted programs for parsing, any ambiguities of the extension with existing forms are automatically resolved - the first defined alternative that matches an input will always be the one recognized. So even if the base language changes such that some or all of the language extension's forms are subsumed by the base language grammar, the extension will continue to be valid.

Paradigm. *Grammar overrides files.* Language extension and dialect files are normally stored in a separate grammar file. Such a grammar modification file is called a *grammar overrides* file, and is included in the TXL program following the include of the base grammar, so that it can refer to the base grammar's defined grammatical types:

```
include "TIL.grm"
include "TILbeginend.grm"
```

For example, while the TIL begin-end extension is independent of the grammatical forms of TIL other than the [statement] form it is extending, in the arrays extension of Figure 12, [expression] and [name] refer to existing grammatical types of TIL.

Paradigm. *Preferential ordering of grammatical forms.* In the begin-end extension the new form is listed as the last alternative, indicating a simple extension that adds to the existing language. When the new forms should be used in preference to existing ones, as in the arrays example, the new form is given as the first alternative and the existing alternatives are listed below, as in the [declaration] and [primary] redefinitions in the arrays extension of TIL:

```
redefine declaration
        'var [name] [opt subscript] ';  [NL]
    |   ...
end redefine

redefine primary
        [name] [opt subscript]
    |   ...
end redefine
```

Because grammatical types are interpreted directly as a parsing program, this means that any input that matches will be parsed as the new form, even if existing old forms would have matched it. So, for example, every var declaration in the arrays extension, including those without a subscript (e.g., "var x;") will be parsed with an optional subscript in the extended language, even though the base grammar for TIL already has a form for it. Similarly, every [name] reference which appears as a [primary] in the extension will be parsed with an [opt subscript] even though there is an existing [name] form for [primary].

Pretty-printing cues for extended forms are specified in redefine statements in the usual way, by adding [NL], [IN] and [EX] output formatting nonterminals to the definitions of the new forms, as in the new [declaration] form above.

Paradigm. *Replacement of grammatical forms.* Grammar type redefinitions can also completely replace the original form in the base grammar. For example, the

[assignment_statement] form of the arrays extension of TIL ignores the definition in the base grammar, and defines its own form which completely replaces it. This means that every occurrence of an [assignment_statement] in the extended language must match the form defined in the dialect.

```
redefine assignment_statement
        [name] [opt subscript] ':= [expression] ';  [NL]
end redefine
```

Paradigm. *Composition of dialects and extensions.* Language extensions and dialects can be composed and combined to create more sophisticated dialects. For example, the module (anonymous class) extension of TIL shown in Figure 12 is itself an extension of the function extension. Extensions are combined by including their grammars in the TXL program in dependency order, for example:

```
include "TIL.grm"
include "TILarrays.grm"
include "TILfunctions.grm"
include "TILmodules.grm"
```

Paradigm. *Modification of existing forms.* Extended forms need not be completely separate alternatives or replacements. When used directly in a redefine rather than as an alternative, the "..." notation still refers to all of the original forms of the nonterminal, modified by the additional syntax around it. For example, in the module extension of TIL (Figure 12), the [function_declaration] form is extended to have an optional *public* keyword preceding it. In this way the module dialect does not depend on what a [function_definition] looks like, only that it exists. Figure 13 shows an example of a program written in the modular TIL language dialect described by the composition of the arrays, functions and modules grammar overrides in Figure 12.

4.4 Robust Parsing

Robust parsing [1] is a general term for grammars and parsers that are insensitive to minor syntax errors and / or sections of code that are unexplained by the input language grammar. Robust parsing is very important in production program analysis and transformation systems since production software languages are often poorly documented, making it difficult to get an accurate grammar, because language compilers and interpreters often allow forms not officially in the language definition, and because languages often have dialects or local custom extensions in practice. For all of these reasons, it is important that analysis and transformation tools such as those implemented in TXL be able to handle exceptions to the grammar so that systems can be at least partially processed.

Paradigm. *Fall-through forms.* The basic paradigm for robust parsing in TXL is to explicitly allow for unexplained input as a dialect of the grammar that adds a last, uninterpreted, alternative to the grammatical forms for which there may be such unofficial or custom variants. For example, we may want to allow for statements that are not in the official grammar by making them the last alternative when we are expecting a statement.

```
File "primes.mtil"

   // this program determines the primes up to maxprimes using the sieve method
    var maxprimes;
    var maxfactor;
    maxprimes := 100;
    maxfactor := 50;        // maxprimes div 2
    var prime;
    var notprime;
    prime := 1;
    notprime := 0;

    module flags
       var flagvector [maxprimes];
       public function flagset (f, tf)
           flagvector [f] := tf;
        end;

      public function flagget (f) : tf
          tf := flagvector [f];
       end;
    end;
    // everything begins as prime
    var i;
    i := 1;
    while i <= maxprimes do
        flagset (i, prime);
        i := i + 1;
    end;
    . . .
```

Fig. 13. Part of an example program in the TIL arrays, functions, modules dialect

Figure 14 shows the grammar overrides for a dialect of TIL that allows for unexplained statement forms. The key idea is that all statements in TIL end with a semicolon - so if we have a form ending in a semicolon that does not match any of the known forms, it must be an unknown statement form. Because alternatives in TXL grammars are tried in order, we can encode this by adding the unknown case as the last form for [statement]:

```
redefine statement
        ...                     % existing forms for [statement]
    |   [unknown_statement]     % fall-through if not recognized
end redefine
```

Paradigm. *Uninterpreted forms.* When parsing, all other alternatives are tested, after which we fall through to the [unknown_statement] form. [unknown_ statement] is any sequence of input items that are not semicolons [not_semicolon*], ended with a semicolon. This ensures that we don't accidentally accept uninterpreted input over a statement boundary.

The [not_semicolon] nonterminal type is the key to flushing uninterpreted input, and uses a standard TXL paradigm for flushing input, [token_or_key]. [token] is a special TXL built-in type that matches any input token that is not a keyword of the grammar, and [key] is a special built-in type that matches any keyword. Thus the following definition describes a type that will accept any single item from the input:

```
define token_or_key
        [token]         % any input token that is not a keyword
    |   [key]           % any keyword
end define
```

Paradigm. *Guarded forms.* In the robust TIL dialect, we must be careful not to throw away a semicolon, and thus we have guarded [token_or_key] with a nonterminal guard. In the [not_semicolon] definition, [not ';] indicates that if the next input token is a semicolon, then we should not accept it as a [token_or_key].

```
define not_semicolon
        [not ';] [token_or_key]    % any item except semicolon
end define
```

[not X] is a generalized grammatical guard that can be used to limit what can be matched by the form following it to those inputs that cannot be recognized as an [X], which can be any grammatical type. Its semantics are simple: if an [X] can be parsed at the current point in the input, then the following form is not tested, otherwise it is. In either case, [not X] does not itself consume any input.

File "TILrobust.grm"

```
% TXL grammar overrides for robust parsing extension of Tiny Imperative Language

% Allow for unrecognized statement forms
redefine statement
        ...                         % refers to all existing forms for [statement]
    |   [unknown_statement]         % add fall-through if we don't recognize a statement
end redefine

define unknown_statement
        [not_semicolon*] ;  [NL]
end define

define not_semicolon
        [not ';] [token_or_key]     % any input item that is not a semicolon
end define

define token_or_key
        [token]                     % any input token that is not a keyword
    |   [key]                       % any keyword
end define
```

Fig. 14. TXL overrides for robust statement parsing in TIL

4.5 Island Grammars

Island grammars [14,19] address a related problem to robust parsing, the problem of embedded code we wish to process in a sea of other text we don't want to process. For example, we may want to analyze only the embedded C code examples in the chapters of a textbook or a set of HTML pages, or only the EXEC SQL blocks in a large set of Cobol programs.

The basic strategy for island grammars in TXL is to invert the robust parsing strategy - we treat the input as a sequence of meaningful things ("islands") and unmeaningful things ('water") (Figure 15). The meaningful things are, for example, TIL programs, and the unmeaningful things are any sequence of input items not beginning with a TIL program.

File "Islands.grm"

```
% Generic grammar for parsing documents
% with embedded islands

% The input is a sequence of interesting
% islands and uninteresting water
redefine program
    [island_or_water*]
end redefine

define island_or_water
        [island]
    |   [water]
end define

% Water is any input that is not an island
define water
    [not_island+]
end define

define not_island
    % any item that does not begin an island
    [not island] [token_or_key]
end define

define token_or_key
        [token]    % any token not a keyword
    |   [key]      % any keyword
end define
```

File "TILislands.txl"

```
% TXL program for parsing documents
% with embedded TIL programs

% Begin with the TIL grammar
include "TIL.grm"

% And the generic island grammar
include "Islands.grm"

% In this case the islands are TIL programs
define island
    [til_program]
end define

define til_program
    [statement+]
end define

% We can now target rules at embedded TIL
% [island]s. But in this case, we just
% delete the non-TIL, to yield code only
rule main
    replace [island_or_water*]
        Water [water]
        Rest [island_or_water*]
    by
        Rest
end rule
```

Fig. 15. TXL generic island grammar (left), and an island parser for embedded TIL programs based on it (right)

Paradigm. *Preferential island parsing.* Figure 15 shows a generic TXL grammar for implementing island grammars to parse documents such as this one, recognizing the embedded islands (such as TIL programs) and ignoring the rest of the text (such as this paragraph). As usual, the trick is that the first alternative [island] is preferred, and the second [water] is tried only if the first fails. Parameterized generic grammars such as this one are frequently used in TXL to encode reusable parsing paradigms such as island grammars.

The generic island grammar is used by defining [island], the interesting form, in the TXL program that includes the generic grammar. The second half of Figure 15 is a TXL program that uses the generic island grammar to make an island grammar for embedded TIL programs in documents such as this tutorial. [island] is defined as [til_program], which uses the included TIL grammar's [statement] form. The analysis or transformation rules can then target the [island] forms only, ignoring the uninterpreted water. In this case, the program simply replaces all occurrences of [water] by the empty sequence, leaving only the embedded TIL programs in the output.

4.6 Agile Parsing

Agile parsing [13] refers to the use of grammar tuning on an individual analysis or transformation task basis. By using the parser to change the parse to better isolate the parts of the program of interest or make them more amenable to the particular transformation or analysis, we can greatly simplify the rules necessary to perform the task.

Paradigm. *Transformation-specific forms.* Agile parsing is implemented in TXL using grammar overrides (redefines) in exactly the same way as we have done for language extensions and dialects. In essence, we create a special dialect grammar for the language in support of the particular task.

The remainder of this paper consists of a sampling of example problems in various applications of source transformation, highlighting the TXL paradigms that are used in each solution.

5 Restructuring Problems

Once we have crafted grammars for our input languages, we can begin using them to support the real work - the transformation and analysis tasks that support software understanding, maintenance, renovation, migration and evolution. The flexibility of the TXL parser is a key to its application in many domains - for example, we exploit agile parsing in many solutions. But the real work is in the transformation and analysis rules.

In the remaining problems from the TXL Cookbook, we concentrate on source code transformation and analysis problems in three categories: restructuring problems, optimization problems, and static and dynamic analysis problems. In each category, we will look at a set of small but real challenges, each couched in terms of TIL and its extensions. We only have space for a few representative examples in each category, chosen not because they are the most useful, but because they introduce new recipes and paradigms.

As we have seen, a TXL "grammar" is not really a grammar - rather it is a functional program for parsing the input, which gives us direct control over the parse, yielding both flexibility and generality. Similarly, a TXL transformation "rule set" is not really a term rewriting system - rather, the rules form a functional program for transforming the input, with similar direct control over tree traversal and strategy, again yielding flexibility and generality.

We begin with problems in basic program restructuring, the heart of applications in refactoring and code improvement. As with our parsing examples, all of our example problems are based on the Tiny Imperative Language (TIL) and its extensions. We will use the grammars and parsing techniques we developed in Section 4 to support all our solutions.

Paradigm. *Programmed functional control.* Transformations and analyses are coded in TXL using *rules* and *functions*. The basic difference between the two is that rules repeatedly search for and transform instances of their pattern until no more can be found, whereas functions transform exactly one instance of their pattern. TXL is a functional language, and the transformation is driven by the application of one rule or function, the *main* rule, to the parse tree of the entire input. All other rules and functions must be explicitly invoked, either in the main rule or in other rules invoked by it.

In contrast to pure term rewriting systems, this functional style gives the programmer fine-grained programmed control over the application of transformation rules on an invocation-by-invocation basis, and tree traversals and strategies can

be customized to each task. Of course, the downside of this flexibility and control is that you *must* do so, the price we pay in TXL for detailed programmability. As we shall see, in practice the common traversals and strategies are simply TXL coding paradigms, which we can learn quickly and reuse as need be. It is these functional paradigms that we will be exploiting in our solutions.

Paradigm. *Transformation scopes.* The result of a TXL rule or function invocation is a transformed copy of the *scope* (parse tree) it is directly applied to. In TXL, scopes of application are explicitly programmed - rules are not global, but transform only the subtree they are applied to. The result of a rule application is (semantically) a completely separate copy from the scope itself - the original TXL variable bound to the scope is unchanged by a rule invocation on it, and retains its original value (parse tree), as in all functional languages. For example, if the TXL variable X is bound to the [number] 1, the rule invocation X [+ 1] yields 2, but does not change X, which retains its original value, 1.

5.1 Feature Reduction

Applications in code analysis and transformation often begin by normalizing the code to reduce the number of features in the code to be analyzed in order to expose basic semantics and reduce the number of cases to analyze. Figure 16 is a simple example of such a feature reduction transformation, the elimination of TIL *for* loops by translation to an equivalent *while*. The transformation has only one rule, [main], which searches for sequences of [statement] beginning with a for statement and replaces the for with an equivalent while statement. While small, this simple example introduces us to a number of TXL paradigms.

It may surprise you to see that the rule is targeted at the type [statement*], a sequence of statements, rather than just [statement], since it is a single *for* statement that we are replacing. The reason for this is that we need to replace the for loop with not one statement but several - the initialization of the iteration variable, the declaration and computation of the upper limit, and the while loop itself. If we had tried to replace a single [statement] with this sequence, we would get a syntax error in the replacement, because TXL rules are constrained to preserve grammatical type in order to guarantee a well-formed result. A sequence of statements [statement*] is not an instance of the type [statement], and thus a replacement of several statements would violate the type constraint.

Paradigm. *Raising the scope of application.* This situation is an example of a general paradigm in TXL - transforming a pattern that is further up the parse tree than what we really want to match, in order to be able to create a result that is significantly different. The saying in TXL is: if you can't create the replacement you want, target further up the tree. In this specific case, we need to create several [statement]s out of one - so we must target the statement sequence [statement*] of which the for statement is a part.

Note that the statements following the for are also captured in the pattern (*MoreStatements*) and preserved in the result. This is a part of the paradigm - if we had not allowed for these, the pattern could match only sequences containing

File "TILfortowhile.txl"

```
% Convert Tiny Imperative Language "for" statements to "while" form

% Based on the TIL grammar
include "TIL.grm"

% Preserve comments in output
#pragma -comment

% Rule to convert every "for" statement
rule main
    % Capture each "for" statement, in its statement sequence context
    % so that we can replace it with multiple statements
    replace [statement*]
        'for Id [id] := Expn1 [expression] 'to Expn2 [expression] 'do
            Statements [statement*]
        'end;
        MoreStatements [statement*]

    % Need a unique new identifier for the upper bound
    construct UpperId [id]
        Id [_ 'upper] [!]

    % Construct the iterator
    construct IterateStatement [statement]
        Id := Id + 1;

    % Replace the whole thing
    by
        'var Id;
        Id := Expn1;
        'var UpperId;
        UpperId := (Expn2) + 1;
        'while Id - UpperId 'do
            Statements [. IterateStatement]
        'end;
        MoreStatements
end rule
```

Fig. 16. TXL transformation to convert for statements to while statements

exactly one statement - that is, the last statement of a sequence. There is no cost to copying these from the pattern to the result, since like many functional languages TXL optimizes flow-through copies.

Paradigm. *Explicit patterns.* The pattern for the for loop is fully explicated, that is, it matches all of its parts right away rather than just a [for_statement] which we could then take apart. Similarly, the replacement contains all the parts of the result explicitly rather than constructing a [while_statement] and replacing it whole. This example-like way of expressing rules is a TXL style - making the pattern and replacement show as much as possible of the form of the actual intended pattern and result target code rather than the constructed terms.

TXL uses the same parser (i.e., the TXL grammar you specify) to parse patterns and replacements in rules as it does for input. Thus it constructs all of the intermediate terms for you. This means that there is no cost to explicating details in a pattern, and it would be no more efficient to have a pattern searching for a [for_statement] only than for the entire pattern we have coded in the [main] rule, because the parsed pattern is in fact a [for_statement] anyway.

Given this preference for an example-like style, it may also surprise you to see that the iteration statement (*IterateStatement*) is separately constructed and appended [.] to the sequence of statements in the body of the loop rather than appearing in the replacement directly. The reason for this is the definition of sequence in TXL - the sequence type [X*] has a recursive definition, deriving [X] [X*] or [empty]. Thus although a statement at the head of a sequence ([statement] [statement*] as in the pattern of this rule) is a valid [statement*], a statement at the end, [statement*] [statement], is not. Therefore the TXL [.] (sequence append) built-in function is provided to allow for this, and the rule uses it to append the new statement to those in the loop body.

It may also surprise you to see that the literal identifiers and keywords in both the pattern and the replacement have been quoted using a single quote ' in all cases. While this is not necessary (except for the TXL keyword "end"), TXL programmers often choose to quote literal identifiers to remind the reader that they are not TXL variable references but part of the output text.

Paradigm. *Generating unique new identifiers.* The rule uses two built-in functions, [_] and [!], to generate a unique new identifier for the introduced upper bound variable. In the construct of *UpperId*, a new identifier is constructed from the original for iteration variable name *Id*, to which the literal identifier "upper" is appended with underscore using the [_] built-in function to form a new identifier (for example, if Id is "i", then we have "i_upper"). The new identifier is then made globally unique using the unique built-in function [!], which appends a number to it to create a new identifier unused anywhere else in the input (for example, "i_upper27").

This first example did its transformations in place - let's look at one that moves things around a bit.

5.2 Declarations-to-Global

One of the standard challenges for transformation tools is the ability to move things about, and in particular to make transformations at an outer level that depend on things deeply embedded in an inner level and vice-versa. In the next two examples, we will look at each of these kinds of problems in turn.

In the first problem, we are simply going to move all declarations in the TIL program to the global scope. Even though TIL declarations seem to be able to appear anywhere according to the TIL grammar, their meaning is apparently global, since no scope rules are defined. In this transformation, we make the true meaning of embedded declarations explicit by promoting all declarations to one global list at the beginning of the program.

The simplest solution to this problem (Figure 17) uses two common paradigms of TXL, *type extraction* and *type filtering*. The basic strategy is shown in the main rule, which has three steps: construct a copy of all the declarations in the program as a sequence, construct a copy of the program with all declarations removed, and concatenate the one to the other to form the result.

```
File "TILtoglobal.txl"

% Make all TIL declarations global

% Based on the TIL base grammar
include "TIL.grm"

% Preserve comments in output
#pragma -comment

% The main rule - in this case a function,
% applies only once

function main
    replace [program]
        Program [statement*]

    % Extract all statements,
    % then filter for declarations only
    construct Declarations [statement*]
        _ [^ Program] [removeNonDeclarations]

    % Make a copy of the program
    % with all declarations removed
    construct ProgramSansDeclarations [statement*]
        Program [removeDeclarations]

    % The result consists of the declarations
    % concatenated with the non-declarations
    by
        Declarations [. ProgramSansDeclarations]
end function

rule removeDeclarations
    % Rule to remove every declaration
    % at every level from statements
    replace [statement*]
        Declaration [declaration]
        FollowingStatements [statement*]
    by
        FollowingStatements
end rule

rule removeNonDeclarations
    % Rule to remove all statements that
    % are not declarations from statements
    replace [statement*]
        NonDeclaration [statement]
        FollowingStatements [statement*]

    % Check the statement isn't a declaration
    deconstruct not NonDeclaration
        _ [declaration]

    % If so, take it out
    by
        FollowingStatements
end rule
```

Fig. 17. TXL transformation to move all declarations to the global scope

Extracting all the declarations from the program is done in two steps, using the extract [^] built-in rule to get a sequence of all the statements of the program, and then removing all those that are not declarations.

```
construct Declarations [statement*]
        _ [^ Program] [removeNonDeclarations]
```

Paradigm. *Extracting all instances of a type.* The extract built-in function [^] is applied to a scope of type [T*] for any type [T], and takes as parameter a bound variable V of any type. The rule constructs a sequence containing a copy of every occurrence of an item of type [T] in V and replaces its scope with the result. In our case, a sequence containing a copy of every [statement] in the program is constructed. Extract ignores its original scope, so it is normally empty to begin with. In this case, we have used the empty variable "_", a special TXL variable denoting an empty item, as the scope of the rule. This is the usual way that extract is used.

Paradigm. *Filtering all instances of a type.* The second step in this construct uses the subrule [removeNonDeclarations] to remove all non-declarations from the constructed sequence of all statements. (The constructor could have extracted all [declaration]s directly, but this would cause problems later when we tried to concatenate them to the beginning of the program.) The subrule uses a common filtering paradigm in TXL, looking for any occurrence of a sequence of statements beginning with a statement that is not a declaration, and replacing

it with the sequence without the beginning statement. The rule continues until it can find no remaining instances in its scope.

Paradigm. *Negative patterns.* Determining that a statement is not a declaration involves another common paradigm in TXL - a negated deconstructor. A normal deconstructor simply matches a bound variable to a pattern for example:

```
deconstruct Statement
      Assignment [assignment_statement]
```

which succeeds and binds Assignment if the [statement] to which Statement is bound consists entirely of an assignment statement.

In this case, however, we are interested in statements that are *not* a [declaration], so we use *deconstruct not* to say that our match succeeds only if the deconstructor fails (that is, the [statement] bound to NonDeclaration is not a [declaration]. Although it has a pattern, a *deconstruct not* does not bind any pattern variables, since to succeed it must not match its pattern. Thus any variable names in the negated deconstructor's pattern are irrelevant, and in this case we have explicitly indicated that by using the anonymous name "_" in the pattern.

```
deconstruct not NonDeclaration
       _ [declaration]
```

The same filtering paradigm is used in the second constructor of the main rule to remove all declarations from the copy of the program used in the result of the rule. This general removal paradigm can be used with any simple, complex or guarded pattern to remove items matching any criterion from a scope.

Finally, the replacement of the rule simply appends the copy of the program without declarations to the extracted declarations, yielding a result with all declarations at the beginning of the program.

5.3 Declarations-to-Local

The other half of the movement challenge is the ability to make transformations on an inner level that depend on things from an outer level. One such problem is localization, in which things at an outer level are to be gathered and moved to an inner level. It can be used to support clustering of related methods, refactoring to infer methods, creation of inferred classes, and so on.

In this next problem, we assume that TIL is a scoped language rather than unscoped. The idea is to find all declarations of variables that are artificially global, and localize them as much as possible to the deepest inner scope in which they are used. In some sense it is the inverse of the previous problem.

Figure 18 shows a TXL solution to this problem. The main rule for this transformation uses two steps - "immediatize" and "localize". The [immediatizeDeclarations] rule moves declarations as far down in their scope as possible, to immediately before the first statement that uses their declared variable.

For example, if we have the scope shown on the left below (a), then the first step, [immediatizeDeclarations], will yield the intermediate result in the middle (b). The second step, [localizeDeclarations], then looks for compound statements

```
File "TILtolocal.txl"

% Move all declarations in a TIL program
% to their most local location

% Based on the TIL base grammar
include "TIL.grm"

% Preserve comments
#pragma -comment

% Transformation to move all declarations
% to their most local location -
% immediately before their first use,
% in the innermost block they can be.

rule main
    % This rule's pattern matches its result,
    % so it has no natural termination point
    replace [program]
        Program [program]

    % So we add an explicit fixed-point
    % guard - after each application of the
    % two transformations, we check to see
    % that something more was actually done
    construct NewProgram [program]
        Program [immediatizeDeclarations]
                [localizeDeclarations]
    deconstruct not NewProgram
            Program
    by
        NewProgram
end rule

rule immediatizeDeclarations
    % Move declarations past statements
    % that don't depend on them.
    % Use a one pass ($) traversal
    replace $ [statement*]
        'var V [id];
        Statement [statement]
        MoreStatements [statement*]

    % We can move the declaration past a
    % statement if the statement does not
    % refer to the declared variable
    deconstruct not * [id] Statement
            V
    by
        Statement
        'var V;
        MoreStatements
end rule

rule localizeDeclarations
    % Move declarations outside a structured
    % statement inside if following statements
    % do not depend on the declared variable.
    % Again, use a one pass ($) traversal
    replace $ [statement*]
        Declaration [declaration]
        CompoundStatement [statement]
        MoreStatements [statement*]

    % Check that it is some kind of compound
    % statement (one with a statement list inside)
    deconstruct * [statement*] CompoundStatement
            _ [statement*]

    % Check that the following statements
    % don't depend on the declaration
    deconstruct * [id] Declaration
            V [id]
    deconstruct not * [id] MoreStatements
            V

    % Alright, we can move it in.
    % Another solution might use agile parsing
    % to abstract all these similar cases into one
    by
        CompoundStatement
            [injectDeclarationWhile Declaration]
            [injectDeclarationFor Declaration]
            [injectDeclarationIfThen Declaration]
            [injectDeclarationIfElse Declaration]
        MoreStatements
end rule

function injectDeclarationWhile
                    Declaration [declaration]
    % There is no legal way that the while
    % Expn can depend on the declaration,
    % since there are no assignments between
    % the declaration and the Expn
    replace [statement]
        'while Expn [expression] 'do
            Statements [statement*]
        'end;
    by
        'while Expn 'do
            Declaration
            Statements
        'end;
end function

  . . .    (other injection rules similar)
```

Fig. 18. TXL transformation to localize all declarations

into which an immediately preceding declaration can be moved, and moves the declaration ("var x;" in the example) inside, yielding the result (c) on the right.

```
var y;
var x;
read y;
y := y + 6;
if y > 10 then
    x := y * 2;
    write x;
end;
```

(a)

```
var y;
read y;
y := y + 6;
var x;
if y > 10 then
    x := y * 2;
    write x;
end;
```

(b)

```
var y;
read y;
y := y + 6;
if y > 10 then
    var x;
    x := y * 2;
    write x;
end;
```

(c)

Paradigm. *Transformation to a fixed point.* Because declarations may be more than one level too global, the process must be repeated on the result until a fixed point is reached. This is encoded in the main rule, which is an instance of the standard fixed-point paradigm for TXL rules.

Although its only purpose is to invoke the other rules, [main] is a rule rather than a function because we expect it to continue to look for more opportunities to transform its result after each application. But unless we check that something was actually done on each application, the rule will never halt since its replacement NewProgram is a [program] and therefore matches its pattern. To terminate the rule, we use a deconstructor as an explicit fixed-point test:

```
deconstruct not NewProgram
        Program
```

The deconstructor simply tests whether the set of rules has changed anything on each repeated application, that is, if the NewProgram is exactly the same as the matched Program. If nothing has changed, we are by definition at a fixed point. This rule is a complete generic paradigm for fixed-point application of any rule set - only the set of rules applied in the constructor changes.

Paradigm. *Dependency sorting.* The rule [immediatizeDeclarations] works by iteratively moving declarations over statements that do not depend on them. In essence, this is a dependency sort of the code. The rule continues to move declarations down until every declaration is immediately before the first statement that uses its declared variable. (This could be done more efficiently by moving declarations directly, but our purpose here is to demonstrate as many paradigms as possible in the clearest and simplest way.) Dependency sorting in this way is a common paradigm in TXL, and we will see it again in other solutions.

Paradigm. *Deep pattern match.* The dependency test uses another common paradigm in TXL - a *deep deconstruct*. This is similar to the negated deconstruct used in the previous problem, but this time we are not just interested in whether Statement does not match something, we are interested in whether it does not *contain* something. Deep deconstructs test for containment by specifying the type of the pattern they are looking for inside the bound variable, and a pattern of that type to find. In this case, we are looking to see if there is an instance of an identifier (type [id]) exactly like the declared one (bound to V).

Paradigm. *One pass rules.* The [immediatizeDeclarations] rule also demonstrates another paradigm of TXL - the "one-pass" rule. If there are two declarations in a row, this rule will contually move them over one another, never coming to a fixed point. For this reason, the rule is marked as one-pass using *replace $*. This means that the scope should be searched in linear fashion for instances of the pattern, and replacements should not be directly reexamined for instances of the pattern. In this case, if we move a declaration over another, we don't try to move the other over it again because we move on to the next sequence of statements in one-pass rather than recursive fashion.

The second rule in this transformation, [localizeDeclarations], looks for instances of a declaration that has been moved to immediately before a compound

statement (such as if, while, for) and checks to see whether it can be moved inside the statement's scope. The rule uses all of the paradigms outlined above - it is one-pass (*replace $*) so that it does not try the same case twice, and it uses deep pattern matching both to get the declared identifier V from the Declaration and to check that the following statements MoreStatements do not depend on the declaration we want to move inside, by searching for uses of V in them.

A new use of deconstruct in this rule is the deep deconstruct of CompoundStatement, which is simply used to check that we actually have an inner scope in the statement in which to move the declaration.

Paradigm. *Multiple transformation cases.* The replacement of this rule demonstrates another paradigm, the programming of cases in TXL. There are several different compound statements into which we can move the declaration: while statements, for statements, then clauses and else clauses. Each one is slightly different, and so they have different patterns and replacements. In TXL such multiple cases use one function for each case, all applied to the same scope.

In essence this is the paradigm for case selection or if-then-else in TXL - application of one function for each case. Only one of the functions will match any particular instance of CompoundStatement, and the others that do not match will leave the scope untouched. TXL functions and rules are *total*, that is, they have a defined result, the identify transformation, when they do not match.

Paradigm. *Context-dependent transformation rules.* In each case, the Declaration to be inserted into the CompoundStatement is passed into the function for the case using a rule parameter. Rule parameters allow us to carry context from outer scopes into rules that transform inner scopes, and this is the paradigm for context-dependent transformation in TXL. In this case we pass the Declaration from the outer scope into the rule that transforms the inner scope.

The context carried in can be arbitrarily large or complex - for example, if the inner transformation rule wanted to change small things inside its scope but depended on global things, we could pass a copy of the entire program into the rule. Outer context can also be passed arbitrarily deeply into subrules, so if a small change deeply inside a sub-sub-subrule depended on something in the outer scope, we could pass a copy all the way in.

5.4 Goto Elimination

The flagship of all restructuring problems is goto elimination - the inference of structured code such as while loops and nested if-then-else from spaghetti-coded goto statements in legacy languages such as Cobol. In this example we imagine a dialect of TIL that has goto statements, and infer equivalent while statements where possible. Figure 19 gives the grammar for a dialect of TIL that adds goto statements and labels, so that we can write programs like the one shown on the left below (a). Our goal is to recognize and transform loop-equivalent goto structures into their while loop form, like the result (b) on the right.

```
// Factor an input number
var n;
var f;
write "Input n please";
read n;
write "The factors of n are";
f := 2;
// Outer loop over potential factors
factors:
    if  n = 1 then
        goto endfactors;
    end;
// Inner loop over multiple instances
// of same factor
multiples:
    if (n / f) * f != n then
        goto endmultiples;
    end;
    write f;
    n := n / f;
    goto multiples;
endmultiples:
    f := f + 1;
    goto factors;
endfactors:
```

(a)

```
// Factor an input number
var n;
var f;
write "Input n please";
read n;
write "The factors of n are";
f := 2;
// Outer loop over potential factors
while n != 1 do
    // Inner loop over multiple instances
    // of same factor
    while (n / f) * f = n do
        write f;
        n := n / f;
   end;
   f := f + 1;
end;
```

(b)

An example TXL solution to the problem of recognizing and transforming while-equivalent goto structures is shown in Figure 19. The basic strategy is to catalogue the patterns of use we observe, encode them as patterns, and use one rule per pattern to replace them with their equivalent loop structures. In practice we would first run a goto normalization (feature reduction) transformation to reduce the number of cases.

The program presently recognizes two cases: "forward' while structures, which begin with an if statement guarding a goto and end with a goto back to the if statement, and "backward" whiles, which begin with a labelled statement and end with an if statement guarding a goto branching back to it.

By now most of the TXL code will be looking pretty familiar. However, this example has two new paradigms to teach us. The first is the match of the pattern in the rule [transformForwardWhile]. Ideally, we are looking for a pattern of the form:

```
replace [statement*]
    L0 [label] ':
        'if C [expression] 'then
            'goto L1 [label] ';
        'end;
        Statements [statement*]
        'goto L0 ';
    L1 ':
        Follow [statement]
    Rest [statement*]
```

Paradigm. *Matching a subsequence.* The trailing Rest [statement*] is necessary since we are matching a subsequence of an entire sequence. If the pattern were to end without including the trailing sequence (i.e., without Rest), then it would only match when the pattern appeared as the last statements in the sequence of statements, which is not what we intend.

File "TILgotos.grm"

```
% Dialect of TIL that adds goto statements

redefine statement
       ...
    |  [labelled_statement]
    |  [goto_statement]
    |  [null_statement]
end redefine

define labelled_statement
    [label] ': [statement]
end define

define goto_statement
    'goto [label] ';  [NL]
end define

% Allow for trailing labels
define null_statement
    [NL]
end define

define label
    [id]
end define

% Add missing "not" operator to TIL
redefine primary
        ...
    |  '! [primary]
end redefine
```

File "TILgotoelim.txl"

```
% Goto elimination in TIL programs

% Recognize and resolve while-equivalent
% goto structures.

% Using the goto dialect of basic TIL
include "TIL.grm"
include "TILgotos.grm"

% Preserve comments in this transformation
#pragma -comment

% Main program - just applies the rules
% for cases we know how to transform.

function main
    replace [program]
        P [program]
    by
        P [transformForwardWhile]
          [transformBackwardWhile]
end function

% Case 1 - structures of the form
%        loop:
%            if Cond then goto endloop; end
%            LoopStatements
%            goto loop;
%        endloop:
%            TrailingStatements

rule transformForwardWhile
    % We're looking for a labelled if guarding
    % a goto - it could be the head of a loop
    replace [statement*]
        L0 [label] ':
            'if C [expression] 'then
         'goto L1 [label] ';
            'end;
        Rest [statement*]
    % If we have a goto back to the labelled if,
    % we have a guarded loop (i.e., a while)
    % The "skipping" makes sure we look only
    % in this statement sequence, not deeper
    skipping [statement]
    deconstruct * Rest
        'goto L0 ';
        L1 ':
            Follow [statement]
        FinalRest [statement*]
    % The body of the loop is the statements
    % after the if and before the goto back
    construct LoopBody [statement*]
        Rest [truncateGoto L0 L1]
    by
        'while '! (C) 'do
            LoopBody
        'end;
        Follow
        FinalRest
end rule

rule transformBackwardWhile

    . . . (similar to above for backward case)

end rule

% Utility rule used by all cases

function truncateGoto L0 [label] L1 [label]
    skipping [statement]
    replace * [statement*]
        'goto L0 ';
        L1 ':
            Follow [statement]
        FinalRest [statement*]
    by
        % nothing
end function
```

Fig. 19. TXL dialect grammar to add goto statements and labels to TIL, and transformation to eliminate gotos (showing first case only)

What is not so obvious is why we could not simply write the pattern above directly in the rule. The reason again has to do with the definition of [X*], which as we recall is recursively defined as [X] [X*] or empty. The pattern above is trying to match [statement] [statement*] [statement] [statement] [statement*], which can't be parsed using that definition no matter how we group it.

Paradigm. *Matching a gapped subsequence.* The TXL paradigm to match such "gapped" sequences is the one used in the [transformForwardWhile] rule. In it, we first match the head of the pattern we are looking for, that is, the leading if statement and the statements following it. We then search in the statements following it for the trailing pattern, the goto back and the ending forward label. The trick of the paradigm is that we must not look inside the statements of the sequence, because we want the trailing pattern to be in the same statement sequence. This is achieved using a *skipping deep deconstruct*.

```
skipping [statement]
deconstruct * Rest
        'goto L0 ';
    L1 ':
        Follow [statement]
        FinalRest [statement*]
```

This deconstructor says that we only have a match if we can find the goto back and the ending forward label without looking inside any of the statements in the sequence (that is, if they are both at the same level, in the statement sequence itself). "skipping [T]" limits a search to the parse tree nodes above any embedded [T]s - in our case, above any statements, so that the goto back is in the same sequence as the heading if statement, completing the pattern we are looking for.

Paradigm. *Truncating the tail of a sequence.* The other new paradigm this example shows us is the truncation of a trailing subsequence, achieved by the function [truncateGoto], which removes everything from the goto on from the statements following the initial if statement. The trick in this function is to look for the pattern heading the trailing subsequence we want to truncate, and replacing it and the following items by an empty sequence. Once again we use the *skipping* notation, since we don't want to accidentally match a similar instance in a deeper statement.

6 Optimization Problems

Source transformation tools are often used in source code optimization tasks of various kinds, and TXL is no exception. In this section we attack some traditional source code optimizations, observing the TXL paradigms that support these kinds of tasks. Once again, our examples are based on the Tiny Imperative Language (TIL) and its extensions.

6.1 Statement-Level Code Motion

The first example problem is on the border between restructuring and optimization: moving invariant assignments and computations out of while loops. In the first solution, we simply look for assignment statements in while loops that are independent of the loop (that is, that don't change over the iterations of the loop). For example, in this loop, the assignment to x does not depend on the loop and can be moved out:

```
var j; var x; var y; var z;
j := 1; x := 5; z := 7;
while j != 100 do
    y := y + j -1;
    x := z * z;
    j := j + 1;
end;
```

Figure 20 shows a solution to this problem for TIL programs. The key to the solution is the function [loopLift], which, given a while loop and an assignment statement in it, checks to see whether the assigned expression of the assignment contains only variables that are not assigned in the loop, and that the assigned variable of the assignment is assigned exactly once in the loop. If both these conditions are met, then the assignment is moved out by putting a copy of it before the loop and deleting it from the loop.

The function uses a number of TXL paradigms. It begins by deconstructing the assignment statement it is passed to get its parts, then uses the extract paradigm to get all of the variable references in the assigned expression. Both of these paradigms we have seen before. The interesting new paradigm is the guarding of the transformation using *where* clauses:

```
% We can only lift the assignment out if all the identifiers in its
% expression are not assigned in the loop ...
where not
    Loop [assigns each IdsInExpression]

% ... and X itself is assigned only once
deconstruct * Body
    X := _ [expression];
    Rest [statement*]
where not
    Rest [assigns X]

% ... and the effect of it does not wrap around the loop
construct PreContext [statement*]
    Body [deleteAssignmentAndRest X]
where not
    PreContext [refers X]
```

Paradigm. *Guarding a transformation with a complex condition.* Where clauses guard the pattern match of a rule or function with conditions that are tested by a subrule or set of subrules. If the *where* clause is positive (i.e., has no *not* modifier), then the subrule must match its pattern for the rule to proceed. If it is a *where not*, as in these cases, then it must not match its pattern.

Paradigm. *Condition rules.* The subrules used in a *where* clause are of a special kind called *condition rules*, which have only a pattern and no replacement. The pattern may be simple, as in the [assigns] and [refers] subrules of this example, which simply check to see if their parameter occurs in the context of their scope, or they may be complex, involving other deconstructors, where clauses and subrules. In either case, a condition subrule simply matches its pattern or not, and the where clause using it succeeds or not depending on whether it matches. If multiple subrules are used in the condition, the where clause succeeds if any one of them matches, and fails only if all do not match (or conversely for *where not*, succeeds only if none match).

```
File "TILcodemotion.txl"

% Lift independent TIL assignments outside
% of while loops

% Based on the TIL grammar
include "TIL.grm"

% Lift all independent assignments out of loops
rule main
    % Find every loop
    replace [statement*]
        while Expn [expression] do
            Body [statement*]
        'end;
        Rest [statement*]

    % Get all the top-level assignments in it
    construct AllAssignments [statement*]
        Body [deleteNonAssignments]

    % Make a copy of the loop to work on
    construct LiftedLoop [statement*]
        while Expn do
            Body
        'end;

    % Only proceed if there are assignments
    % left that can be lifted out.
    % The [?loopLift] form tests if the
    % [loopLift] rule can be matched -
    % "each AllAssignments" tests this
    % for any of the top-level internal
    % assignments
    where
        LiftedLoop
            [?loopLift Body each AllAssignments]

    % If the above guard succeeds,
    % some can be moved out, so go ahead
    % and move them, replacing the original
    % loop with the result
    by
        LiftedLoop
            [loopLift Body each AllAssignments]
            [. Rest]
end rule

% Attempt to lift a given assignment
% outside the loop
function loopLift Body [statement*]
                  Assignment [statement]
    deconstruct Assignment
        X [id] := E [expression];

    % Extract a list of all the identifiers
    % used in the expression
    construct IdsInExpression [id*]
        _ [^ E]

    % Replace the loop and its contents
    replace [statement*]
        Loop [statement*]

    % We can only lift the assignment out
    % if all the identifiers in its
    % expression are not assigned in the loop ...
    where not
        Loop [assigns each IdsInExpression]

    % ... and X itself is assigned only once
    deconstruct * Body
        X := _ [expression];
        Rest [statement*]
    where not
        Rest [assigns X]

    % ... and the effect of it
    % does not wrap around the loop
    construct PreContext [statement*]
        Body [deleteAssignmentAndRest X]
    where not
        PreContext [refers X]

    % Now lift out the assignment
    by
        Assignment
        Loop [deleteAssignment Assignment]
end function

% Utility rules used above

% Delete a given assignment from a scope
function deleteAssignment Assignment [statement]
    replace * [statement*]
        Assignment
        Rest [statement*]
    by
        Rest
end function

% Delete all non-assignments in a scope
rule deleteNonAssignments
    replace [statement*]
        S [statement]
        Rest [statement*]
    deconstruct not S
        _ [assignment_statement]
    by
        Rest
end rule

% Delete everything in a scope from
% the first assignment to X on
function deleteAssignmentAndRest X [id]
    replace * [statement*]
        X := E [expression];
        Rest [statement*]
    by
        % nada
end function

% Does a scope assign to the identifier?
function assigns Id [id]
    match * [assignment_statement]
        Id := Expn [expression];
end function

% Does a scope refer to the identifier?
function refers Id [id]
    match * [id]
        Id
end function
```

Fig. 20. TXL transformation to lift independent assignments out of while loops

Paradigm. *Each element of a sequence.* The first where condition in the [loopLift] function also uses another paradigm - the *each* modifier.

```
where not
    Loop [assigns each IdsInExpression]
```

each takes a sequence of type [X*] for any type [X], and calls the subrule once with each element of the sequence as parameter. So for example, if IdsInExpression is bound to the sequence of identifiers "a b c", then "where not Loop [assigns each IdsInExpression]" means "where not Loop [assigns 'a] [assigns 'b] [assigns 'c]", and the guard succeeds only if none of the [assigns] calls matches its pattern. This is a common TXL paradigm for checking multiple items at once.

The main rule in this example simply finds every while loop, extracts all the assignment statements in it by making a copy of the statements in the loop body and deleting those that are not assignments, and then calls [loopLift] with *each* to try to move each of them outside the loop. Rather than use the fixed-point paradigm, this main rule uses a *where* clause as a guard to check whether there are any assignments to move in advance. To do this, it actually uses the [loopLift] function itself to check - by converting it to a condition using [?].

```
where
    LiftedLoop [?loopLift Body each AllAssignments]
```

Paradigm. *Using a transformation rule as a condition.* [?loopLift] means that [loopLift] should not do any replacement - rather, it should act as a condition rule, simply checking whether its complex pattern matches or not. Thus the where clause above simply checks whether [loopLift] will succeed for any of the assignments, and the rule only proceeds if at least one will match.

6.2 Common Subexpression Elimination

Common subexpression elimination is a traditional optimization transformation that searches for repeated subexpressions whose value cannot have changed between two instances. The idea is to introduce a new temporary variable to hold the value of the subexpression and replace all instances with a reference to the temporary. For example, if the input contains the code on the left (a) below, then the output should be the code (b) shown on the right.

```
var a; var b;
read a;
b := a * (a + 1);
var i;
i := 7;
c :=  a * (a + 1);
```

(a)

```
var a; var b;
read a;
var t;
t := a * (a + 1);
b := t;
var i;
i := 7;
c := t;
```

(b)

A TXL solution to this problem for TIL programs is shown in Figure 21. The solution uses a number of new paradigms for us to look at. To begin, the program uses *agile parsing* to modify the TIL grammar in two ways.

Paradigm. *Grammatical form abstraction.* First, it overrides the definition of [statement] to gather all compound statements into one statement type. This

File "TILcommonsubexp.txl"

```
% Recognize and optimize common subexpressions

% Based on the TIL base grammar
include "TIL.grm"

% Preserve comments
#pragma -comment

% Override to abstract compound statements
redefine statement
        [compound_statement]
    |   ...
end redefine

define compound_statement
        [if_statement]
    |   [while_statement]
    |   [for_statement]
end define

% Allow statements to be attributed
% so we don't mistake one we've
% generated for one we need
% to process
redefine statement
        ...
    |   [statement] [attr 'NEW]
end redefine

% Main rule
rule main
    replace [statement*]
        S1 [statement]
        SS [statement*]
    % Don't process statements we generated
    deconstruct not * [attr 'NEW] S1
        'NEW
    % We're looking for an expression ...
    deconstruct * [expression] S1
        E [expression]
    % ... that is nontrivial ...
    deconstruct not  E
        _ [primary]
    % ... and repeated
    deconstruct * [expression] SS
        E
    % See if we can abstract it
    % (checks if variables assigned between)
    where
        SS [?replaceExpnCopies S1 E 'T]
    % If so, generate a new temp name ...
    construct T [id]
        _ [+ "temp"] [!]
    % ... declare it, assign it the expression,
    % and replace instances with it
    by
        'var T;    'NEW
        T := E;    'NEW
        S1 [replaceExpn E T]
        SS [replaceExpnCopies S1 E T]
end rule

% Recursively replace copies of a given
% expression with a given temp variable id,
% provided the variables used in the
% expression are not assigned in between

function replaceExpnCopies S1 [statement]
                  E [expression] T [id]
    construct Eids [id*]
        _ [^ E]

    % If the previous statement did not assign
    % any of the variables in the expression
    where not
        S1 [assigns each Eids]

    % Then we can continue to substitute the
    % temporary variable for the expression
    % in the next statement ...
    replace [statement*]
        S [statement]
        SS [statement*]

    % ... as long as it isn't a compound
    % statement that internally assigs one of
    % the variables in the expression
    where not all
        S [assignsOne Eids]
          [isCompoundStatement]
    by
        S [replaceExpn E T]
        SS [replaceExpnCopies S E T]
end function

% Check to see if a statement assigns
% any of a list of variables

function assignsOne Eids [id*]
    match [statement]
        S [statement]
    where
        S [assigns each Eids]
end function

function assigns Id [id]
    match * [statement]
        Id := _ [expression] ;
end function

function isCompoundStatement
    match [statement]
        _ [compound_statement]
end function

rule replaceExpn E [expression] T [id]
    replace [expression]
        E
    by
        T
end rule
```

Fig. 21. TXL transformation to recognize and optimize common subexpressions

redefinition takes advantage of TXL's programmed parsing to prefer that if, while and for statements be parsed as [compound_statement]. The original forms are still in the definition of [statement] (denoted by "..."), but since our new form appears first, all of them will be parsed as [compound_statement]. This paradigm is often used to gather forms so that we can use one rule to target all of the forms at once rather than having several rules for the different grammatical types.

Paradigm. *Marking using attributes.* The second technique used here is grammar attributes, denoted by the [attr] modifier. TXL grammar attributes denote optional parts of the grammar that will not appear in the unparsed output text. They can be of any grammatical type, including complex types with lots of information in them. In this case, the attribute is simply the identifier "NEW", and it is added to allow us to mark statements that are inserted by the transformation so that we don't mistake them for a statement to be processed.

Marking things that have been generated or already processed using attributes is a common technique in TXL, and is often the easiest way to distinguish things that have been processed from those that have not. The new attributed form is recursive, allowing any statement to be marked as "NEW".

The main rule finds any statement containing a nontrivial expression, determined by deconstructing it to ensure that it is not simply a [primary]. It then deconstructs the following statements to determine if the expression is repeated in them. If so, then it uses the conditional guard paradigm to check that the repetition will be legally transformable [?replaceExpnCopies]. A new unique temporary name of the form "temp27" is then created using the unique identifier paradigm, and finally, statements are generated to declare and assign the expression to the new temporary.

This is where the NEW attribute comes in. By marking the newly generated statements with the NEW attribute, we are sure that they will not be matched by the main rule and reprocessed. The remainder of the replacement copies the original statement and following statements, substituting the new temporary name for the expression in the original statement [replaceExpn E T] and any subsequent uses in following statements [replaceExpnCopies S1 E T].

Paradigm. *Tail-recursive continuation.* Rule [replaceExpnCopies] (Figure 21) introduces us to another new paradigm - continuing a transformation through a sequence as long as some condition holds. In this case, we can continue to substitute the temporary name for the common expression as long as the variables in the expression are not assigned to.

In TXL such situations are encoded as a tail-recursive function, which processes each statement one by one checking that the conditions still hold, until it fails and terminates the recursion. In each recursion we pass the previous statement as parameter, and first check that it has not assigned any of the identifiers used in the expression, again using the where-not-each paradigm of the previous problem. We then match the next statement in the sequence, and check that it is not a compound statement that assigns any of the identifiers in the expression.

Paradigm. *Guarding with multiple conditions.* This check uses a new paradigm - *where not all*. As we've seen in previous paradigms, where clauses normally check whether *any* of the condition rules matches. When *all* is specified, the check is whether *all* of the condition rules match. Thus the where clause here:

```
where not all
    S [assignsOne Eids]
      [isCompoundStatement]
```

checks whether it is both the case that one of the identifiers used in the expression is assigned by the statement, and that the statement is a compound statement (in which case our simple algorithm choose to give up and stop).

If the check succeeds and either the statement is not a compound statement or does not assign any of the variables in the original expression, then instances of the expression are substituted in the matched statement and we recursively move on to the next one.

6.3 Constant Folding

Constant folding, or optimizing by recognizing and precomputing compile-time known subexpressions, is another traditional optimization technique. In essence, the solution is a partial evaluation of the program, replacing named constants by their values and interpreting resulting operations on those values. Thus a constant folding algorithm must have rules to evaluate much of the expression sublanguage of the target language.

The solution for TIL (Figure 22) is in two parts: recognition and substitution of constant assignments to variables that are not destroyed, and interpretation of constant subexpressions. Of course, these two processes interact, because substitution of constant values for variables yields more constant subexpressions to evaluate, and evaluation of constant subexpressions yields more constant values for variables. So in the main rule we see the now familiar paradigm for a fixed point, continuing until neither rule changes anything.

The [propagateConstants] rule handles the first half of the problem, searching for assignments of constant values to variables (e.g., "x := 5;") that are not destroyed by a subsequent assignment in the same statement sequence. The two deep deconstructs of Rest are the key to the rule. The first one ensures that the following statements do not subsequently assign to the variable, destroying its constant value. The second one makes sure that there is a reference to the variable to substitute. When both conditions are met, the value is substituted for all references to the variable in the following statements.

The second half of the transformation is the interpretation of constant subexpressions (possibly created by the first half substituting constant variable values). The rule [foldConstantExpressions] simply applies a set of rules each of which knows how to evaluate an operator with constant operands. In addition to the simple cases, a number of special cases, such as multiplying any expression by zero, are also handled. [foldConstantExpressions] continues applying the set of evaluation rules until none of them changes anything and a fixed point is reached.

```
File "TILconst.txl"

% Constant propagation and folding for TIL

% Begin with the TIL base grammar
include "TIL.grm"

% Preserve comments in this transformation
#pragma -comment

% Main function
rule main
    replace [program]
        P [program]
    construct NewP [program]
        P [propagateConstants]
          [foldConstantExpressions]
    deconstruct not NewP
        P
    by
        NewP
end rule

% Constant propagation - find each
% constant assignment to a variable,
% and if it is not assigned again then
% replace references with the constant
rule propagateConstants
    replace [statement*]
        Var [id] := Const [literal] ;
        Rest [statement*]
    deconstruct not * [statement] Rest
        Var := _ [expression] ;
    deconstruct * [primary] Rest
        Var
    by
        Var := Const;
        Rest [replaceExpn Var Const]
end rule

rule replaceExpn Var [id] Const [literal]
    replace [primary]
        Var
    by
        Const
end rule

% Constant folding - find and evaluate
% constant expressions
rule foldConstantExpressions
    replace [expression]
        E [expression]
    construct NewE [expression]
        E % Generic folding of pure
          % constant expressions
          [resolveAddition]
          [resolveSubtraction]
          [resolveMultiplication]
          [resolveDivision]
          % Other special cases
          [resolveAdd0]
          [resolveSubtract0]
          [resolveMultiply1Right]
          [resolveMultiply1Left]
          [resolveParentheses]
    % Continue until we don't
    % find anything to fold
    deconstruct not NewE
        E
    by
        NewE
end rule

% Utility rules to do the arithmetic
rule resolveAddition
    replace [expression]
        N1 [integernumber]
           + N2 [integernumber]
    by
        N1 [+ N2]
end rule

rule resolveSubtraction
    replace [expression]
        N1 [integernumber]
           - N2 [integernumber]
    by
        N1 [- N2]
end rule

% ... other operator folding rules
  . . .
```

Fig. 22. TXL transformation to fold constant subexpressions

6.4 Statement Folding

Our last optimization example is statement folding, the elimination of statements that cannot be reached because the conditions that guard them are known at compile time, for example, when an if condition is known to be true or false. In practice, constant folding and statement folding go together - constant folding precomputes conditional expressions, some of which are then known to be true or false, allowing for statement folding. These problems are closely related to conditional compilation. Transformations to implement preprocessors and conditional compilation are essentially the same as constant and statement folding.

Figure 23 shows a TXL solution to the statement folding problem for TIL if and while statements with known conditions. In this case the main rule is a

```
File "TILstmtfold.txl"

% Statement folding using TIL

% Look for opportunities to reduce code
% footprint by optimizing out unreachable code

% Begin with the TIL base grammar
include "TIL.grm"

% Preserve comments
#pragma -comment

 % Main function
function main
    replace [program]
        P [program]
    by
        P [foldTrueIfStatements]
          [foldFalseIfStatements]
end function

% Folding rules for constant condition ifs

rule foldTrueIfStatements
    % Find an if statement
    replace [statement*]
        'if Cond [expression] 'then
            TrueStatements [statement*]
            ElseClause [opt else_statement]
        'end;
        Rest [statement*]

    % with a constant true condition
    where
        Cond [isTrueEqual] [isTrueNotEqual]

    % and replace it with the true part
    by
        '// Folded true if
        TrueStatements [. Rest]
end rule

rule foldFalseIfStatements
    % Find an if statement
    replace [statement*]
        'if Cond [expression] 'then
            TrueStatements [statement*]
            ElseClause [opt else_statement]
        'end;
        Rest [statement*]

    % with a constant false condition
    where not
        Cond [isTrueEqual]
             [isTrueNotEqual]

    % and replace it with the false part
    construct FalseStatements [statement*]
        _ [getElseStatements ElseClause]
    by
        '// Folded false if
        FalseStatements [. Rest]
end rule

function getElseStatements
            ElseClause [opt else_statement]
    deconstruct ElseClause
        'else
            FalseStatements [statement*]
    replace [statement*]
        % default none
    by
        FalseStatements
end function

% Utility functions to detect statically
% true conditions - these can be as
% smart as we wish

  . . .
```

Fig. 23. TXL transformation to fold known if statements

function, since none of the rules changes anything that may create new instances of the others, and thus the fixed point paradigm is not needed.

Paradigm. *Handling optional parts.* In the false if condition case (rule [foldFalseIfStatements]) there is a new paradigm used to get the FalseStatements from the else clause of the if statement. Beginning with an empty sequence using the empty variable "_", a separate function is used to get the FalseStatements from the else clause. The reason for this construction is that the [else_statement] is optional - there may not be one. So beginning with the assumption there is none (i.e., the empty sequence) we used the [getElseStatements] function to both check if there is one (by deconstructing the ElseClause) and if so to replace the empty sequence by the FalseStatements.

Paradigm. *Creating output comments.* Both cases illustrate another TXL paradigm - the creation of target language comments. Besides explicitly marking identifiers intended to be literal output, quoting of items in TXL marks something to be lexically interpreted in the target language rather than TXL. Thus

a target language comment can be created in a TXL replacement simply by prequoting it (Figure 23, rule [foldFalseIfStmts]). This can be handy when marking sections of code that have been transformed in output.

7 Static and Dynamic Analysis Problems

Now that we've tried some of the simpler problems and introduced many of the standard paradigms for using TXL, it's time to attack some more realistic challenges. Static and dynamic analysis tasks, including program comprehension, security analysis, aspect mining and other analyses, are commonly approached using parsing and source transformation tools. In this section we demonstrate the use of TXL in several example static and dynamic analyses, including static metrics, dynamic tracing, type inference, slicing, clone detection, code markup, unique renaming and fact extraction.

7.1 Program Statistics

In our first analysis example, we demonstrate TXL's use in computing static program metrics. Figure 24 shows a TXL program designed to gather statement usage statistics for TIL programs. The input is any TIL program, and the output is empty. However, the program uses the TXL message built-in functions to print out several statement statistics about the program on the standard error stream as it matches the measured features. The error stream output of this program when processing the "factors.til" program of Figure 7 looks like this:

```
Total: 11
Declarations: 2
Assignments: 3
Ifs: 0
Whiles: 2
Fors: 0
Reads: 1
Writes: 3
```

The program uses the TXL type extraction paradigm that we have seen before to collect all statements of each type into sequences, and then counts them using the sequence [length] built-in function to give the statistic.

Paradigm. *Counting feature instances.* This is a general paradigm that when combined with agile parsing (to gather our desired grammatical forms) and the filtering paradigm we have seen previously (to refine to the exact subset we are interested in) can be used to count instances of any feature or pattern in the program, including most standard static metrics. An important point here is the use of the empty variable "_" as the [number] scope of the counting constructors. When used in a [number] context, the empty variable plays the role of the value zero, which is often a good place to start a numeric computation.

Paradigm. *Dynamic error stream output.* The program uses the TXL error stream built-in function [putp] to output messages reporting the statistics as they are computed. [putp] is modeled after the C *printf* function, and acts in

```
File "TILstats.txl"

% Gather TIL statement statistics

% Begin with the TIL base grammar
include "TIL.grm"

% Compute and output statement kind counts,
% and replace program with an empty one.
% There are many different ways to do this -
% this naive way is simple and obvioulsy
% correct, but exposes TXL's need for generics.
% Another less clear solution could use
% polymorphism to avoid the repetition.

function main
    replace [program]
        Program [program]

    % Count each kind of statement we're
    % interested in by extracting all of
    % each kind from the program

    construct Statements [statement*]
        _ [^ Program]
    construct StatementCount [number]
        _ [length Statements]
          [putp "Total: %"]

    construct Declarations [declaration*]
        _ [^ Program]
    construct DeclarationsCount [number]
        _ [length Declarations]
          [putp "Declarations: %"]

    construct Assignments [assignment_statement*]
        _ [^ Program]
    construct AssignmentsCount [number]
        _ [length Assignments]
          [putp "Assignments: %"]

    construct Ifs [if_statement*]
        _ [^ Program]
    construct IfCount [number]
        _ [length Ifs] [putp "Ifs: %"]

    construct Whiles [while_statement*]
        _ [^ Program]
    construct WhileCount [number]
        _ [length Whiles] [putp "Whiles: %"]

    construct Fors [for_statement*]
        _ [^ Program]
    construct ForCount [number]
        _ [length Fors] [putp "Fors: %"]

    construct Reads [read_statement*]
        _ [^ Program]
    construct ReadCount [number]
        _ [length Reads] [putp "Reads: %"]

    construct Writes [write_statement*]
        _ [^ Program]
    construct WriteCount [number]
        _ [length Writes] [putp "Writes: %"]
    by
        % nothing
end function
```

Fig. 24. TXL transformation to collect and report statement statistics

much the same way, printing out its string parameter with the output text of the scope it is applied to substituted for the "%" in the string. In this case, the scope is a [number], and the corresponding number value is printed in the message.

In general, the scope of [putp] can be any type at all, and both [putp] and its simpler form [put], which takes no parameter and simply prints out the text of its scope, can be used to instrument and debug TXL programs as they execute.

7.2 Self Tracing Programs

The addition of auxiliary monitoring code to a program is a common transformation task, and in this example we demonstrate the paradigms for adding such code using TXL. The problem is to transform a TIL program to a self-tracing version of itself, one that prints out each statement just before it is executed. This is a model for a large number of transformations used in instrumentation and dynamic analysis tasks such as test coverage and dynamic flow analysis.

To distinguish statements that are generated or have already been processed by the transformation, the program uses the same attribute marking paradigm we have seen before to mark statements that have been generated or already processed, in this case marking with the attribute "TRACED".

Paradigm. *Eliding detail.* In order that we don't print out entire multi-line messages for compound statements, the program also allows for an elision marker

```
File "TILtrace.txl"

% Make a TIL program self-tracing

% Replaces every statement with a write
% statement of its text followed by itself

% Begin with the TIL base grammar
include "TIL.grm"

% Don't bother preserving comments,
% we only want to run the result

% Pragma to tell TXL that our string
% escape convention uses backslash
#pragma -esc "¨

% Allow elided structured statements
redefine statement
        ...
    |   [SP] '... [SP]
end redefine

% Allow for traced statements - the TRACED
% attribute marks statements already done
redefine statement
        ...
    |   [traced_statement]
end redefine

define traced_statement
    [statement] [attr 'TRACED]
end define

% Main rule
rule main
    % Result has two statements where one
    % was before, so work on the sequence
    replace [statement*]
        S [statement]
        Rest [statement*]
    % Semantic guard: if it's already
    % done don't do it again
    deconstruct not S
        _ [statement] 'TRACED
    % Make a concise version of
    % structured statements
    construct ConciseS [statement]
        S [deleteBody]
    % Get text of the concise statement
    construct QuotedS [stringlit]
        _ [+ "Trace: "] [quote ConciseS]
    by
        'write QuotedS;    'TRACED
        S                  'TRACED
        Rest
end rule

% Utility function - replace the body
% of a structured statement with ...
function deleteBody
    replace * [statement*]
        _ [statement*]
    by
        '...
end function
```

Fig. 25. TXL transformation to transform TIL program to self-tracing

"..." as a [statement]. This is used in the function [deleteBody], which makes a copy of a statement in which the body has been replaced by "..." so that the trace will be more terse. The function uses a deep search (*replace **) to find the outermost sequence of statements embedded in the statement which is its scope.

The main rule does all of the work, searching for every statement that has not yet been transformed (i.e., that is not yet marked with the attribute TRACED) and inserting a *write* statement to print out its quoted text before it is executed. Both the write statement and the original are attributed with TRACED in the replacement so that they are not themselves transformed again.

Once again we see the paradigm for replacing one element of a sequence with more than one by targeting the higher level sequence [statement*] rather than the element [statement] - and again the pattern and replacement of the rule must preserve the Rest of the statements following the one we are transforming.

Paradigm. *Converting program text to strings.* The construction of the quoted string version of the statement's text to be printed in the trace uses the TXL string manipulation built-in functions [+] and [quote]. Beginning with an empty [stringlit], once again denoted by the empty variable "_", the constructor concatenates the string literal "Trace: " to the quoted text of the statement.

```
construct QuotedS [stringlit]
    _ [+ "Trace: "] [quote ConciseS]
```

The [quote] built-in function creates a string literal containing the text of its parameter, which may be any grammatical type, and concatenates it to its scope, in this case the string "Trace: ". The output of a run of the traced version of the "factors.til" TIL program of Figure 7, when executed, looks like this:

```
Trace: var n;
Trace: write "Input n please";
Input n please
Trace: read n;
read: 6
Trace: write "The factors of n are";
The factors of n are
Trace: var f;
Trace: f := 2;
Trace: while n != 1 do ... end;
Trace: while (n / f) * f = n do ... end;
Trace: write f;
2
Trace: n := n / f;
Trace: f := f + 1;
Trace: while (n / f) * f = n do ... end;
Trace: write f;
3
```

7.3 Type Inference

Calculation of derived or inferred attributes of items in a program is a common analysis task, forming part of type checking, optimization of dynamically typed programs, translation between languages, and business type analysis such as the Y2K problem. In this example we demonstrate the TXL paradigms for concrete and abstract type inference, using a transformation to infer static types for the untyped variables in a TIL program from the contexts in which they are used.

TIL declares untyped variables, originally intended to be all integer. However, the addition of string values to the language led to string variables, making the language effectively dynamically typed. However, perhaps TIL variables could be statically typed if they are used consistently. This transformation infers the type of every variable in a TIL program from its uses and explicitly adds types to declarations using the new form: "var x: integer;" where the valid types are "integer" and "string". Variables of inconsistent type are flagged as an error.

Figure 26 shows a solution to this problem. Using the precedence (PRIORITY) version of the TIL grammar, the program begins with several grammar overrides. First, the new form of declarations is added, by allowing for an optional type specification on each variable declaration. Types "int", "string" and "UNKNOWN" are allowed. The special type UNKNOWN is included so that we can mark variables whose type is inconsistent or that we cannot infer, so that error messages can be generated when we are done.

Next, we override the definition of [primary] to allow for a type attribute on every variable reference, literal constant and parenthesized expression in the program. We will use these attributes to record local inferences we make about types of variables and expressions.

Finally, to make the transformation more convenient, we use agile parsing to make the grammar easier to deal with for this particular problem. Everywhere in the TIL grammar where a variable appears as an [id] (i.e., "left hand side" references outside of expressions), we allow instead a [primary], so that it can be type attributed in the same way as in expressions.

```
redefine assignment_statement
   [primary] ':= [expression] ';   [NL]
end redefine
```

File "TILtypeinfer.txl"

```
% Infer types for variables and expressions

% Infer all expression and variable types,
% add types to variable declarations,
% and flag type conflicts.

% Based on the TIL base grammar
include "TIL.grm"

% Preserve comments
#pragma -comment

% Allow type specs on declarations.

redefine declaration
   'var [primary]
        [opt colon_type_spec] '; [NL]
end redefine

define colon_type_spec
   ': [type_spec]
end define

define type_spec
   'int | 'string | 'UNKNOWN
end define

% Allow type attributes on primaries.

redefine primary
   [subprimary] [attr type_attr]
end redefine

define subprimary
   [id] | [literal] | '( [expression] ')
end define

define type_attr
   '{ [opt type_spec] '}
end define

% Conflate all [id] refs to [primary],
% to make attribution rules simpler.

redefine assignment_statement
   [primary] ':= [expression] ';   [NL]
end redefine

redefine for_statement
    'for [primary] := [expression]
            'to [expression] 'do   [IN][NL]
       [statement*]                [EX]
    'end ';                        [NL]
end redefine

redefine read_statement
   'read [primary] ';  [NL]
end redefine

% The typing process has several steps:

%    1. introduce complete parenthesization,
%    2. enter default empty type attributes,
%    3. attribute literal expressions,
%    4. infer attributes from context until
%       a fixed point is reached,
%    5. set type attribute of uninferred
%       items to UNKNOWN,
%    6. add declaration types from variables'
%       inferred type attribute,
%    7. report errors (i.e., UNKNOWN types),
%    8. undo complete parenthesization.

function main
    replace [program]
        P [program]
    by
        P [bracket]
          [enterDefaultAttributes]
          [attributeStringConstants]
          [attributeIntConstants]
          [attributeProgramToFixedPoint]
          [completeWithUnknown]
          [typeDeclarations]
          [reportErrors]
          [unbracket]
end function

% Rules to introduce and undo complete
% parenthesization to allow for detailed
% unambiguous type attribution

function bracket
    replace [program]
        P [program]
    by
        P [bracketExpressions]
          [bracketComparisons]
          [bracketTerms] [bracketFactors]
end function

rule bracketExpressions
    skipping [expression]
    replace [expression]
        E [expression] Op [logop] C [comparison]
    by
       '( E [bracketExpressions]
         Op C [bracketExpressions] ')
end rule

 . . . (bracketComparisons, bracketTerms,
        bracketFactors similar)

function unbracket
    replace [program]
        P [program]
    by
        P [unbracketExpressions]
          [unbracketComparisons]
          [unbracketTerms] [unbracketFactors]
end function

rule unbracketExpressions
    replace [expression]
        '( E [expression] ')
            { Type [type_spec] }
    by
        E
end rule

 . . . (unbracketComparisons, unbracketTerms,
        unbracketFactors similar)

% Rule to add empty type attributes
% to every primary expression and variable

rule enterDefaultAttributes
    replace [attr type_attr]
    by
        { }
end rule
```

Fig. 26. TXL transformation to infer types of TIL variables

```
% The meat of the type inference algorithm.
% Infer empty type attributes from the types
% in the context in which they are used.
% Continue until no more can be inferred.

rule attributeProgramToFixedPoint
    replace [program]
        P [program]
    construct NP [program]
        P [attributeAssignments]
          [attributeExpressions]
          [attributeComparisons]
          [attributeTerms]
          [attributeFactors]
          [attributeForIds]
          [attributeDeclarations]
    deconstruct not NP
         P
    by
         NP
end rule

rule attributeStringConstants
    replace [primary]
        S [stringlit] { }
    by
        S { string }
end rule

rule attributeIntConstants
    replace [primary]
        I [integernumber] { }
    by
        I { int }
end rule

rule attributeAssignments
    replace [assignment_statement]
        X [id] { } := SP [subprimary]
            {Type [type_spec] };
    by
        X { Type } := SP { Type };
end rule

. . . (attributeForIds similar)

rule attributeExpressions
   replace [primary]
      ( P1 [subprimary] {Type [type_spec]}
        Op [logop] P2 [subprimary] {Type} ) { }
   by
      ( P1 {Type} Op P2 {Type} ) {Type}
end rule

. . . (attributeComparisons, attributeTerms,
       attributeFactors similar)

rule attributeDeclarations
    replace [statement*]
       'var Id [id] { } ;
        S [statement*]
    deconstruct * [primary] S
        Id { Type [type_spec] }
    by
       'var Id { Type };
        S [attributeReferences Id Type]
end rule

rule attributeReferences
        Id [id] Type [type_spec]
    replace [primary]
        Id { }
    by
        Id { Type }
end rule

% Once a fixed point has been reached,
% set all such remaining empty type
% attributes to UNKNOWN.

rule completeWithUnknown
    replace [attr type_attr]
       { }
    by
       { UNKNOWN }
end rule

% Add an explicit type to every untyped
% variable declaration, from the
% variable's inferred type attribute.

rule typeDeclarations
    replace [declaration]
        'var Id [id] { Type [type_spec] };
    by
        'var Id { Type } : Type;
end rule

% Report type errors.  An UNKNOWN
% attribute indicates either a conflict or
% not enough information to infer a type.

rule reportErrors
    replace $ [statement]
       S [statement]
    skipping [statement*]
    deconstruct * [type_spec] S
        'UNKNOWN

    % Issue an error message.
    % [pragma "-attr"] allows attributes
    % to be printed in the message.

    construct Message [statement]
        S [pragma "-attr"] [message
    "*** ERROR: Unable to resolve types in:"]
          [stripBody] [putp "%"]
          [pragma "-noattr"]
    by
         S
end rule

function stripBody
    replace * [statement*]
       _ [statement*]
    by
        % nothing
end function
```

Fig. 27. TXL transformation to infer types of TIL variables (*continued*)

```
redefine for_statement
    'for [primary] := [expression] 'to [expression] 'do  [IN][NL]
       [statement*]                                      [EX]
    'end                                                 [NL]
end redefine

redefine read_statement
   'read [primary] ';  [NL]
end redefine
```

Paradigm. *Grammatical form generalization.* Technically this allows for many forms that are not legal in TIL - for example, the form "4 := 7;". But since this is a program analysis transformation, we can assume that our input is well-formed. This is a general paradigm in TXL - using a more lenient grammar than the target language in order to subsume forms that will be handled in the same way into a single grammatical type in order to simplify transformation rules. This is the core idea in agile parsing [13].

The transformation rules use a number of new paradigms: normalization of the program so that all cases are the same, inference of attributes to a fixed point using a set of local inference rules, promotion of locally inferred attributes to the global scope, and denormalization of the final result.

Paradigm. *Program normalization.* In this case the normalization is simple - the normalizing rule [bracketExpressions] converts every [expression] in the program to a fully parenthesized version. Full parenthesization both makes every expression into a [primary], which allows it to be attributed with a type due to the overrides above, and limits every [expression] to one operator, since subexpression operands will be also be fully parenthesized. This reduces our inference problem to only one case - that of a single operator, simplifying and clarifying the inference rules. This kind of simplifying normalization is typical of many source analysis tasks, and is essential to any complex inference problem in TXL.

The denormalizing rule [unbracketExpressions] both unparenthesizes and removes the inferred type attribute of the expression, since the result of type inference is in the explicit types on variable declarations in the result.

Paradigm. *Default analysis results.* Following normalization, a default empty type attribute is added to every [primary] in the input program using the rule [enterDefaultAttributes]. This secondary normalization again reduces the number of cases, since rules can handle both attributed and unattributed primaries in the same way. Such defaulting is also typical of inference tasks in TXL.

Once these normalizations are complete, the actual type inference algorithm is simple - we just look for opportunities to infer the type of as yet untyped items in a context where other types are known. This begins with simple typing of literal primaries, whose type is native to their value, using the rules [attributeStringConstants] and [attributeIntConstants]. This is the base case of the inductive inference algorithm.

Paradigm. *Inductive transformation.* The process then proceeds using a small set of contextual inference rules, using the fixed-point paradigm to halt when not more types can be inferred. The key rule is [attributeOperations], which infers

the type of an operator expression from the types of its operands, which looks for operations with two operands of the same type, and infers that the result must also be of that type. Of course, such inference rules depend on the programming language, but the basic strategy remains the same.

Another key inference rule is [attributeDeclarations], which infers the type of a variable declaration from any one of its references, and then marks all other references with the same type. [attributeDeclarations] uses a deep deconstructor of the statements following the declaration to see if a type has been inferred for any reference to the variable, and if so, gives the declaration that type and marks all other references in the following statements with it. (Using the local-to-global paradigm we saw in the restructuring examples.) This new typing can in turn can give more information for the next iteration of the operator inference rule above, and so on. Once the inference rules have come to a fixed point, any remaining unknown types are given the special type UNKNOWN.

Finally, we insert the inferred types into all variable declarations, and then report errors by printing out all statements containing types we could not infer - those attributed as UNKNOWN.

```
construct Message [statement]
    S [pragma "-attr"]
      [message "*** ERROR: Unable to resolve types in:"] [stripBody] [putp "%"]
      [pragma "-noattr"]
```

Paradigm. *Making attributes visible.* The message constructor illustrates the ability of TXL to include attributes in the output text - by turning on the "-attr" option, attributes are printed in the output text, in this case of the [putp] function, so that they can be seen in the error message:

```
"*** ERROR: Unable to resolve types in:
x {UNKNOWN} := ( y {string} + 1 {int} ) {UNKNOWN};
```

The [pragma] function allows us to turn TXL options on and off dynamically.

7.4 Static Slicing

Dependency analysis is an important and common static analysis technique, and one of the most common dependency analyses is the static slice. As defined by Weiser [22], a (backward) *slice* is the executable subset of the statements in a program that preserves the behavior observable at a particular statement. If the slice is executed with the same input as the program, then all variable values when the slice reaches the statement will be the same as if the original program were to be executed to the same point. Often the value of one particular variable is designated as the one of interest, in which case values of others can be ignored.

Slicing algorithms are usually carried out by building a dependency graph for the program and then using graph algorithms to reduce it to the slice, which is mapped back to source statements afterward. However, as we have seen in the type inference example, in TXL we can compute dependency chains directly, using the inductive transformation paradigm.

Figure 28 shows a TXL program for backward slicing of TIL programs. The program uses a related TXL paradigm called *cascaded markup*, in which, beginning with one statement marked as the one of interest, statements which directly

influence that statement are marked, and then those that influence those statements, and so on until a fixed point is reached.

The program begins with grammar overrides to allow for XML-like markup of TIL statements. The input to the program will have one such statement marked as the one of interest, as shown on the left (a) below. The output slice for this input is shown on the right (b).

```
        var chars;
        var n;
        read n;
        var eof_flag;
        read eof_flag;
        chars := n;
        var lines;
        lines := 0;
        while eof_flag do
            lines := lines + 1;
            read n;
            read eof_flag;
            chars := chars + n;
        end;
        write (lines);
<mark> write (chars); </mark>
```

(a)

```
var chars;
var n;
read n;
var eof_flag;
read eof_flag;
chars := n;


while eof_flag do

    read n;
    read eof_flag;
    chars := chars + n;
end;

write (chars);
```

(b)

Here the statement "write (chars);" has been marked. The challenge for the slicer is to trace dependencies backwards in the program to mark only those statements that can influence the marked one, yielding the backward slice for the program (b).

Paradigm. *Cascaded markup.* The basic strategy is simple: an assignment to a variable is in the backward slice if any subsequent use of the variable is already in the slice. The rule that implements the strategy is [backPropogateAssignments] (Figure 28). We have previously seen the "skipping" paradigm - here it prevents us from remarking statements inside an already marked statement.

The other markup propagation rules are simply special cases of this basic rule that propagate markup backwards into loop and if statements and around loops, and out to containing statements when an inner statement is marked. The whole set of markup propagation rules is controlled by the usual fixed-point paradigm that detects when no more propagation can be done.

Once a fixed point is reached, the program simply removes all unmarked statements [removeUnmarkedStatements] and unused declarations [removeRedundantDeclarations], then removes all markup to yield the program slice. The result fot the example (a) above is shown on the right (b). (Line spacing is shown to align with the original code, and is not part of the output.)

7.5 Clone Detection

Clone detection is a popular and interesting source analysis problem with a wide range of applications, including code reduction and refactoring. In this problem, we demonstrate the basic techniques for clone detection using TXL. Clone detection can vary in granularity from statements to functions or classes.

```
File "TILbackslice.txl"

% Backward static slicing of TIL programs

% Backward slice from a statement marked up
% using <mark> </mark>

% Begin with the TIL base grammar
include "TIL.grm"

% Allow for XML markup of TIL statements
redefine statement
        ...
    |   [marked_statement]
end redefine

define marked_statement
            [xmltag] [statement] [xmlend]
end define

define xmltag
        < [SPOFF] [id] > [SPON]
end define

define xmlend
        < [SPOFF] / [id] > [SPON]
end define

% Conflate while and for statements
% into one form to optimize handling
% of both forms in one rule
redefine statement
        [loop_statement]
    |   ...
end redefine

define loop_statement
        [loop_head]                 [NL][IN]
            [statement*]            [EX]
        'end;                       [NL]
end define

define loop_head
        while [expression] do
    |   for [id] := [expression]
            to [expression] do
end define

% The main function gathers the steps
% of the transformation: induce markup
% to a fixed point, remove unmarked
% statements, remove declarations for
% variables not used in the slice,
% and strip markups to yield the
% sliced program

function main
    replace [program]
        P [program]
    by
        P [propagateMarkupToFixedPoint]
          [removeUnmarkedStatements]
          [removeRedundantDeclarations]
          [stripMarkup]
end function

% Back propagate markup of statements
% beginning with the initially marked
% statement of interest.
% Continue until a fixed point

rule propagateMarkupToFixedPoint
    replace [program]
        P [program]

    construct NP [program]
        P [backPropogateAssignments]
          [backPropogateReads]
          [whilePropogateControlVariables]
          [loopPropogateMarkup]
          [loopPropogateMarkupIn]
          [ifPropogateMarkupIn]
          [compoundPropogateMarkupOut]

    % We're at a fixed point when P = NP
    deconstruct not NP
        P
    by
        NP
end rule

% Rule to back-propagate markup of
% assignments.  A previous assignment is
% in the slice if its assigned variable
% is used in a following marked statement

rule backPropogateAssignments
    skipping [marked_statement]
    replace [statement*]
        X [id] := E [expression] ;
        More [statement*]
    where
        More [hasMarkedUse X]
    by
        <mark> X := E; </mark>
        More
end rule

% Similar rule for read statements

rule backPropogateReads
    skipping [marked_statement]
    replace [statement*]
        read X [id] ;
        More [statement*]
    where
        More [hasMarkedUse X]
    by
        <mark> read X; </mark>
        More
end rule

function hasMarkedUse X [id]
    match * [marked_statement]
        M [marked_statement]
    deconstruct * [expression] M
        E [expression]
    deconstruct * [id] E
        X
end function

% Other propagation rules for loops
% and compound statements

  . . .
```

Fig. 28. TXL transformation to compute a backward slice

Since TIL does not have functions or classes, we are using structured statements (if, for, while) as a simple example. While this is clearly not a realistic case, detection of function or block clones (in any language) would be very similar.

Figure 29 shows a TXL solution to the detection of structured statement clones in TIL. The program begins with a set of grammar overrides to gather all of the structured statements into one type so that we don't need separate rules for each kind. As with the backward slicing example, we use XML-like markup to mark the results of our analysis. In this case, we want to mark up all instances of the same statement as members of the same clone class, so we allow for an XML attribute in the tags.

Paradigm. *Precise control of output spacing.* These overrides illustrate another output formatting cue that we have not seen before - the explicit control of output spacing. TXL normally uses a set of default output spacing rules that insert spaces around operators and other special symbols such as "<". Unfortunately, these spacing rules lead to strange output in the case of XML markup - for example, the XML tag "<clone class=4>" would be output as "< clone class = 4 >", which is not even legal XML.

The TXL built-in types [SPOFF], [SPON] and [SP] used here allow the programmer to take complete control of output spacing. Like [NL], [IN] and [EX], none of these has any effect on input parsing. [SPOFF] temporarily turns off TXL's output spacing rules, and [SPON] restores them. Between the two, items will be output with no spacing at all. The [SP] type allows programmers to insert spacing as they see fit, in this case forcing a space between the tag identifier and the attribute identifier in output tags.

As we have seen is often the case, the main TXL program works in two stages. In the first stage, a sequence containing one instance of each the cloned compound statements in the program is constructed using the function [findStructuredStatementClones]. In the second stage, all instances of each one of these are marked up in XML as instances of that clone class, assigning class numbers as we go (function [markCloneInstances]).

Paradigm. *Context-dependent rules.* The function [findStructuredStatementClones] works by using the subrule [addIfClone] to examine each of the set of all structured statements in the program (StructuredStatements) to see if it appears more than once, and if so adds it to its scope, which begins empty. While we have seen most of this paradigm before, we have not before seen a case where the transformation rule needs to look at both a local item and its entire global context at the same time to determine if it applies.

This kind of global contextual dependency is implemented in TXL using rule parameters. In this case an entire separate copy of StructuredStatements is passed to [addIfClone] so that it can use the global context in its transformation, in this case simply to check if each particular statement it is considering appears twice. This is an instance of the general TXL paradigm for context-dependent transformations, which allows for arbitrary contextual information, including if necessary a copy of the entire original program, to be passed in to a

File "TILclonesexact.txl"

```
% Clone detection for TIL programs

% Find exact clones of structured statements,
% and output the program with clones marked
% up to indicate their clone class.

% Begin with the TIL base grammar
include "TIL.grm"

% We are NOT interested in comments

% Overrides to conflate all structured
% statements into one nonterminal type.

redefine statement
        [structured_statement]
    |   ...
end redefine

define structured_statement
        [if_statement]
    |   [for_statement]
    |   [while_statement]
end define

% Allow XML markup of statements.

redefine statement
        ...
    |   [marked_statement]
end redefine

define marked_statement
    [xmltag]              [NL][IN]
        [statement]       [EX]
    [xmlend]              [NL]
end define

% [SPOFF] and [SPON] temporarily disable
% default TXL output spacing in tags

define xmltag
    < [SPOFF] [id] [SP] [id] = [number] > [SPON]
end define

define xmlend
    < [SPOFF] / [id] > [SPON]
end define

% Main program

function main
    replace [program]
        P [program]

    % First make a table of all repeated
    % structured statements
    construct StructuredClones
              [structured_statement*]
        _ [findStructuredStatementClones P]

    % Mark up all instances of each of them.
    % CloneNumber keeps track of the index of
    % each in the table as we step through it
    export CloneNumber [number] 0
    by
        P [markCloneInstances
              each StructuredClones]
end function

% We make a table of the cloned structured
% statements by first making a table
% of all structured statements in the program,
% then looking for repeats

function findStructuredStatementClones
         P [program]
    % Extract a list of all structured
    % statements in the program
    construct StructuredStatements
              [structured_statement*]
        _ [^ P]
    % Add each one that is repeated
    % to the table of clones
    replace [structured_statement*]
        % empty to begin with
    by
        _ [addIfClone StructuredStatements
                  each StructuredStatements]
end function

function addIfClone
        StructuredStatements [structured_statement*]
        Stmt [structured_statement]
    % A structured statement is cloned if it
    % appears twice in the list of all statements
    deconstruct * StructuredStatements
        Stmt
        Rest [structured_statement*]
    deconstruct * [structured_statement] Rest
        Stmt

    % If it does appear (at least) twice,
    % add it to the table of clones
    replace [structured_statement*]
        StructuredClones [structured_statement*]
    % Make sure it's not already in the table
    deconstruct not * [structured_statement]
            StructuredClones
        Stmt
    by
        StructuredClones [. Stmt]
end function

% Once we have the table of all clones,
% we mark up each instance of each of them
% in the program with its clone class,
% that is, the index of it in the clone table

rule markCloneInstances
        StructuredClone [structured_statement]
    % Keep track of the index of this clone
    % in the table
    import CloneNumber [number]
    export CloneNumber
        CloneNumber [+ 1]

    % Mark all instances of it in the program
    % 'skipping' avoids marking twice
    skipping [marked_statement]
    replace [statement]
        StructuredClone
    by
        <clone class=CloneNumber>
            StructuredClone
        </clone>
end rule
```

Fig. 29. TXL transformation to detect exact structured statement clones in TIL

local transformation. There it can be used both in conditions to guard the local transformation, as is the case this time, or as a source of additional parts to be used in the result of the local transformation.

Paradigm. *Accumulating multiple results.* [addIfClone] also demonstrates another common paradigm - the accumulation of results into a single sequence. Beginning with an empty sequence using the empty variable "_" in the replacement of [findStructuredStatementClones], [addIfClone] adds each result it finds to the end of its scope sequence. It makes sure that it does not put the same statement in twice using a guarding deconstructor, which checks to see if the cloned statement is already in the list:

```
deconstruct not * [structured_statement] StructuredClones
        Stmt
```

Once [findStructuredStatements] has constructed a unique list of all of the cloned structured statements in the program, [markCloneInstances] marks up all of the instances of each one in the program. Each is assigned a unique class number to identify it with its instances using the global variable CloneNumber, which begins at 0 and is incremented by [markCloneInstances] on each call.

Paradigm. *Updating global state.* While TXL is primarily a pure functional language, global state is sometimes required in complex transformations. For this purpose TXL allows global variables, which can be of any grammatical type (including forms that are not in the input language). In this case the global variable CloneNumber is a simple [number] that begins with the value 0. Inside a TXL rule, globals are simply normal local TXL variables. But they can be "exported" to the global scope where their value can be "imported" into another rule where they once again act as a local variable of the rule. Within a rule, the value bound to an imported global is set when it is imported, as if it were bound in a pattern match. The value bound to the variable can only be changed if the rule re-imports or exports the global with a new value.

In this case, on each invocation, [markCloneInstances] imports CloneNumber and immediately constructs and exports a new value for it, the previous value plus one. This new value is used in the replacement of the rule to mark up every instance of the current clone with that clone class number, making it clear which marked up statements are clones of one another in the result.

Of course, exact clone detection is the simplest case, and although interesting, not very realistic. Fortunately, we are using TXL, so modifying our clone detector to handle more aggressive techniques is not difficult. In particular, we can make the clones identifier-independent, like CCFinder [18], just by adding a normalization rule to make all identifiers the same when comparing:

```
rule normalizeIdentifiers
    replace $ [id]
        _ [id]
    by
        'X
end rule
```

If we want to be more precise, we can compare with consistent renaming - that is, where identifiers are normalized consistently with their original names.

```
% Rule to normalize structured statements
% by consistent renaming of identifiers
% to normal form (x1, x2, x3, ...)

rule renameStructuredStatement
    % For each outer structured statement
    % in the scope
    skipping [structured_statement]
    replace $ [structured_statement]
        Stmt [structured_statement]

    % Make a list of all of the unique
    % identifiers in the statement
    construct Ids [id*]
        _ [^ Stmt] [removeDuplicateIds]

    % Make normalized new names of the
    % form xN for each of them
    construct GenIds [id*]
        Ids [genIds 0]

    % Consistently replace each instance
    % of each one by its normalized form
    by
        Stmt [$ each Ids GenIds]
end rule
```

```
% Utility rule -
% remove duplicate ids from a list

rule removeDuplicateIds
    replace [id*]
        Id [id] Rest [id*]
    deconstruct * [id] Rest
        Id
    by
        Rest
end rule

% Utility rule -
% make a normalized id of the form xN
% for each unique id in a list

function genIds NM1 [number]
    % For each id in the list
    replace [id*]
        _ [id]
        Rest [id*]

    % Generate the next xN id
    construct N [number]
        NM1 [+ 1]
    construct GenId [id]
        _ [+ 'x] [+ N]

    % Replace the id with the generated one
    % and recursively do the next one
    by
        GenId
        Rest [genIds N]
end function
```

Fig. 30. TXL rule to consistently normalize identifiers in a TIL statement

Figure 30 shows a TXL rule to consistently rename the identifiers in a TIL structured statement. The rule works by extracting an ordered list of all of the identifiers used in the structured statement, and then generates a list of identifiers of the form x1, x2, x3 and so on of the same length by recursively replacing each identifier in a copy of the list with a generated one.

The result lists might look like this:

```
Ids          xyz  abc  n    wid  zoo
GenIds       x1   x2   x3   x4   x5
```

Paradigm. *Each corresponding pair.* The actual transformation of the original ids to the generated ones is done using the built-in rule [$], which is TXL shorthand for a fast global substitute. The rule application uses a paired *each* to pass the substitute rule each pair of corresponding identifiers in the lists, that is, $[' xyz' x1], [$ 'abc 'x2], and so on. This general paradigm can be used to match any two sequences of corresponding items, for example formals and actuals when analyzing function calls.

7.6 Unique Renaming

Unique renaming [15] gives scope-independent names to all declared items in a program. Unique naming flattens the name space so that every item declared in a program can be unambiguously referred to independent of its context. In

particular, unique naming is useful when creating a relationship database for the program in the form of facts, as in Rigi's RSF [21] or Holt's TA [16] format.

In this example transformation (Figure 31), we uniquely rename all declared variables, functions and modules in programs written in the module extension of TIL to reflect their complete scope. For example, a variable named X declared in function F of module M is renamed M.F.X .

This transformation demonstrates a number of new paradigms. Most obvious is that this process must be done from the innermost scopes to the outermost, so that when renaming things declared in a module M, all of the things declared in an embedded function F have already been renamed F.X. That way, we can simply rename everything transitively inside M with M. to reflect its scope, for example yielding M.F.X .

Paradigm. *Bottom-up traversal.* The paradigm for applying rules "inside out" (from the bottom up, from a parse tree point of view) is used in the main rule of this transformation, [uniqueRename] (Figure 31). [uniqueRename]'s real purpose is to find each declaration or statement that forms a scope, to get its declared name (ScopeName) and then use the [uniqueRenameScope] subrule to rename every declaration in the scope with the ScopeName. But in order for this to work correctly, it must handle the scopes from the inside out (bottom-up).

Bottom-up traversal is done by recursively applying the rule to each matched Scope more deeply before calling [uniqueRenameScope] for the current scope. The paradigm consists of two parts: "skipping [statement]" in [uniqueRename] assures that we go down only one level at a time, and the call to [uniqueRenameDeeper], which simply recursively applies [uniqueRename] to the inside of the current scope, ensures that we process deeper levels before we call [uniqueRenameScope] for the current level. This paradigm is generic and can be used whenever inner elements should be processed before outer.

The actual renaming is done by the rule [uniqueRenameScope], which finds every embedded declaration in a scope (no matter how deeply embedded), and renames both the declaration and all of its references in the scope to begin with the given ScopeName. For example, if ScopeName is M and some inner declaration is so far named F.G.X, then both the declaration and all references to it get renamed as M.F.G.X . Since we are processing inside out, there is no ambiguity with deeper declarations whose scopes have already been processed.

Paradigm. *Abstracted patterns.* [uniqueRenameScope] demonstrates another new paradigm: abstracted matching. Even though the real pattern it is looking for is a Declaration followed by its RestOfScope, the rule matches less precisely and uses a deconstructor to check for the pattern. This is because the replacement will have to consistently change both the Declaration and the RestOfScope in the same way (i.e. renaming occurrences of the declared name). By matching the part that requires change in one piece, the transformation requires only one use of the renaming substitution rule [$], making the rule simpler and clearer.

Once all declarations and embedded references have been renamed, there are two remaining tasks: renaming references to a module's public functions that

File "TILuniquerename.txl"

```
% Uniquely rename every Modular TIL variable
% and function with respect to its context.
% e.g., variable V declared in a while
% statement in function F of module M
% is renamed as M.F.whileN.V

% Begin with the MTIL grammar
include "TIL.grm"
include "TILarrays.grm"
include "TILfunctions.grm"
include "TILmodules.grm"

% Allow for unique names - in this case,
% TIL does not have a field selection operator,
% so we can use X.Y notation for scoped names.
redefine name
        [id]
    |   [id] . [name]
end redefine

% Main program
function main
    replace [program]
        P [program]
    by
        P [uniqueRenameDeeper]
          [uniqueRenameScope 'MAIN]
          [renameModulePublicReferences]
          [renameFunctionFormalParameters]
end function

rule uniqueRename
    % Do each statement on each level once
    skipping [statement]
    replace $ [statement]
        Scope [statement]
    % Only interested in statements with scopes
    deconstruct * [statement*] Scope
        _ [statement*]
    % Use the function, module or unique
    % structure name for it
    construct ScopeName [id]
        _ [makeKeyName Scope]
          [getDeclaredName Scope]
    % Visit inner scopes first, then this one
    by
        Scope [uniqueRenameDeeper]
              [uniqueRenameScope ScopeName]
end rule

% Recursively implement bottom-up renaming
function uniqueRenameDeeper
    replace * [statement*]
        EmbeddedStatements [statement*]
    by
        EmbeddedStatements [uniqueRename]
end function

% Make an identifier for the scope -
% if a declaration, use the declared id,
% otherwise synthesize a unique id from the
% statement keyword
function makeKeyName Scope [statement]
    deconstruct * [key] Scope
        Key [key]
    construct KeyId [id]
        _ [+ Key] [!]
    replace [id]
        _ [id]
    by
        KeyId
end function

function getDeclaredName Scope [statement]
    replace [id]
        _ [id]
    deconstruct Scope
        DeclaredScope [declaration]
    deconstruct * [id] DeclaredScope
        ScopeName [id]
    by
        ScopeName
end function

% Do the actual work - rename each declaration
% and its references with the scope id
rule uniqueRenameScope ScopeName [id]
    % Find a declaration in the scope
    replace $ [statement*]
        DeclScope [statement*]
    deconstruct DeclScope
        Declaration [declaration]
        RestOfScope [statement*]
    % Get its original id
    deconstruct * [name] Declaration
        Name [name]
    % Add the scope id to its name
    construct UniqueName [name]
        ScopeName '. Name
    % Rename the declaration and all
    % references in the scope.
    by
        DeclScope [$ Name UniqueName]
end rule

% This section handles the problem of
% references to a public function outside
% of the module it is declared in

rule renameModulePublicReferences
    % Find a module and its scope
    replace $ [statement*]
        'module ModuleName [name]
            ModuleStatements [statement*]
        'end ;
        RestOfScope [statement*]
    % Get all its public function names
    construct
            UniquePublicFunctionNames [name*]
        _ [extractPublicFunctionName
            each ModuleStatements]
    % Rename all references in the outer scope
    by
        'module ModuleName
            ModuleStatements
        'end ;
        RestOfScope
             [updatePublicFunctionCall
              each UniquePublicFunctionNames]
end rule
```

Fig. 31. TXL transformation to uniquely rename all declared items in TIL programs to reflect their scope

```
function extractPublicFunctionName
          Statement [statement]
    % We're interested only in functions
    deconstruct Statement
        Function [function_definition]
    % Which are public
    deconstruct * [opt 'public] Function
        'public
    % Get the function id
    deconstruct * [name] Function
        UniquePublicFunctionName [name]
    % Add it to the end of the list
    replace * [name*]
    by
        UniquePublicFunctionName
end function
```

```
rule updatePublicFunctionCall
        UniquePublicFunctionName [name]
    % Get the original name
    deconstruct * [name]
            UniquePublicFunctionName
        PublicFunctionName [id]
    % Replace all uses with unique name
    skipping [name]
    replace $ [name]
        PublicFunctionName
    by
        UniquePublicFunctionName
end rule

% Rules to rename function formal parameters
    ... (similar to rules above)
```

Fig. 32. TXL transformation to uniquely rename all declared items in TIL programs to reflect their scope (continued)

are outside its inner scope, and renaming formal parameters. Both of these pose a new kind of transformation problem: how to do a transformation on an outer level of the parse that depends on information from an inner level? Such a transformation is called a local-to-global transformation, and is a standard challenge for source transformation systems.

The TXL solution is demonstrated by the rule [renameModulePublicReferences] in Figure 32. We need to make a transformation of all references to the original name of any public function of the module that occur in the scope in which the module is declared, that is, in RestOfScope. But the public functions of the module cannot be in the pattern of the rule - what to do?

Paradigm. *Inner context-dependent transformation.* The answer is to contextualize the transformation by raising the information we need from the inner scope to the level we are at. In this case, that is done by the construct of UniquePublicFunctionNames, which uses the subrule [extractPublicFunctionName] to get a copy of the unique name of every public function declared in the module. Once we have brought the context up to the level we are at, we can do the transformation we need using [updatePublicFunctionCall] by passing it each public function unique name.

In general, the inner context to be raised could be much deeper or more complex than simply public functions declared one level down. Using a constructor and subrule to bring deeper context up, we can always get what is needed.

7.7 Design Recovery

Design recovery, or fact generation, is the extraction of basic program entities and relationships into an external graph or database that can be explored using graph and relationship analysis tools such as CrocoPat [4], Grok [17], or Prolog. In this problem, we show how TXL can be used to extract facts from programs using source transformation.

File "TILgeneratefacts.txl"

```
% Design recovery (fact extraction) for MTIL

% Given a uniquely renamed MTIL program,
% infer and generate architecture design facts
% contains(), calls(), reads(), writes()

% Begin with the MTIL grammar
include "TIL.grm"
include "TILarrays.grm"
include "TILfunctions.grm"
include "TILmodules.grm"

% Our input has been uniquely renamed by
% TILuniquerename.txl using X.Y notation
redefine name
        [id]
    |   [id] . [name]
end redefine

% Grammar for Prolog facts
include "Facts.grm"

% Override to allow facts on any statement
redefine statement
        ...
    |   ';     % null statement,
               % so we can add facts anywhere
end redefine

% Override to allow facts on any statement
redefine statement
        [fact*] ...
end redefine

% Override to allow facts on any expression
redefine primary
        [fact*] ...
end redefine

% Our output is the facts alone
redefine program
        ...
    |   [fact*]
end redefine

% Main program
function main
    replace [program]
        P [program]
    construct ProgramName [name]
        'MAIN
    construct AnnotatedP [program]
        P [addContainsFacts ProgramName]
          [inferContains]
          [addCallsFacts ProgramName]
          [inferCalls]
          [addReadsFacts ProgramName]
          [inferReads]
          [addWritesFacts ProgramName]
          [inferWrites]
    construct Facts [fact*]
        _ [^ AnnotatedP]
    by
        Facts
end function

% Infer contains() relationships
rule inferContains
    replace $ [declaration]
        ScopeDecl [declaration]
    deconstruct * [statement*] ScopeDecl
        Statements [statement*]
    deconstruct * [name] ScopeDecl
        ScopeName [name]
    by
        ScopeDecl
            [addContainsFacts ScopeName]
            [addContainsParameters ScopeName]
end rule

rule addContainsFacts ScopeName [name]
    skipping [statement]
    replace $ [statement]
        Facts [fact*] Declaration [declaration]
    deconstruct * [name] Declaration
        DeclName [name]
    construct NewFacts [fact*]
        'contains '( ScopeName, DeclName ')
        Facts
    by
        NewFacts Declaration
end rule

function addContainsParameters ScopeName [name]
    replace [declaration]
        Public [opt 'public]
        'function Fname [name]
            '( ParameterNames [name,] )
 OptResultParameter [opt colon_id]
            Statements [statement*]
        'end;
    construct OptResultParameterName [name*]
        _ [getResultParameterName
               OptResultParameter]
    construct ParameterContainsFacts [fact*]
        _ [makeFact 'contains ScopeName
                    each ParameterNames]
          [makeFact 'contains ScopeName
                    each OptResultParameterName]
    construct FactsStatement [statement]
        ParameterContainsFacts ';
    by
        Public
        'function Fname '( ParameterNames )
        OptResultParameter
            FactsStatement
            Statements
        'end;
end function

function getResultParameterName
         OptResultParameter [opt colon_id]
    deconstruct OptResultParameter
        ': ResultParameterName [name]
    replace [name*]
    by
        ResultParameterName
end function
```

Fig. 33. TXL transformation to generate basic facts for an MTIL program

```
% Infer calls() relationships
rule inferCalls
    replace $ [declaration]
        ScopeDecl [declaration]
    deconstruct * [statement*] ScopeDecl
        Statements [statement*]
    deconstruct * [name] ScopeDecl
        ScopeName [name]
    by
        ScopeDecl [addCallsFacts ScopeName]
end rule

rule addCallsFacts ScopeName [name]
    skipping [declaration]
    replace $ [statement]
        Facts [fact*]
        CallStatement [call_statement]
    skipping [id_assign]
    deconstruct * [name] CallStatement
        CalledName [name]
    by
        'calls '( ScopeName, CalledName ')
        Facts
        CallStatement
end rule

% Infer reads() relationships
rule inferReads
    replace $ [declaration]
        ScopeDecl [declaration]
    deconstruct * [statement*] ScopeDecl
        Statements [statement*]
    deconstruct * [name] ScopeDecl
        ScopeName [name]
    by
        ScopeDecl [addReadsFacts ScopeName]
end rule

rule addReadsFacts ScopeName [name]
    skipping [statement]
    replace $ [statement]
        Statement [statement]
    by
        Statement
            [addExpressionReadsFacts ScopeName]
end rule

rule addExpressionReadsFacts ScopeName [name]
    skipping [declaration]
    replace $ [primary]
        Primary [primary]
    deconstruct * [name] Primary
        FetchedName [name]
    construct ReadsFact [fact*]
        'reads ( ScopeName, FetchedName )
    by
        Primary [addFacts ReadsFact]
end rule

% Infer writes() relationships
. . .  ( similar to reads() )

% Utility functions
function makeFact FactId [id]
                  Name1 [name] Name2 [name]
    replace * [fact*]
    by
        FactId ( Name1, Name2 )
end function

function addFacts NewFacts [fact*]
    replace * [fact*]
        Facts [fact*]
    by
        Facts [. NewFacts]
end function
```

Fig. 34. TXL transformation to generate basic facts for an MTIL program (continued)

Figure 33 shows a program that extracts basic structural and usage facts for programs written in the module dialect of TIL. Facts extracted include *contains(), calls(), reads()* and *writes()* relationships for all modules and functions.

Paradigm. *Local fact annotation.* The basic strategy of the program is to annotate the program with facts directly in the local contexts where the fact can be inferred. For example, for the statement: x.y.z := a.b.c; appearing in function M.F, we will annotate the statement with the facts:

```
writes (M.F, x.y.z)
reads (M.F, a.b.c)
x.y.z := a.b.c;
```

In this way we can use local transformations to create the facts where the evidence for them occurs. The actual rules to infer each kind of fact are fairly simple: for each declaration in a scope, we annotate with a *contains()* fact. For each reference to a name in a scope, we annotate with a *reads()* fact. And so on. If more information is needed to infer a fact, we can leverage all of the previous techniques we have seen to assist us: context-dependent transformation, inner context-dependent transformation, bottom-up traversal, and any others we need.

Once the program is completely annotated with facts, the only remaining task is to gather them together, which is done using the usual type extraction paradigm to bring all the facts into one sequence, which we can then output as the result of our fact generation transformation. The final result of this program is a fact base in Prolog form, that looks like this:

```
contains (MAIN, MAIN.maxprimes)
contains (MAIN, MAIN.maxfactor)
writes (MAIN, MAIN.maxprimes)
writes (MAIN, MAIN.maxfactor)
contains (MAIN, MAIN.prime)
writes (MAIN, MAIN.prime)
contains (MAIN, MAIN.flags)
contains (MAIN.flags, MAIN.flags.flagvector)
contains (MAIN.flags, MAIN.flags.flagset)
contains (MAIN.flags.flagset, MAIN.flags.flagset.f)
writes (MAIN.flags.flagset, MAIN.flags.flagvector)
reads (MAIN.flags.flagset, MAIN.flags.flagset.f)
. . .
```

8 Conclusion and Future Work

The TXL Cookbook is very much a work in progress, and what we have seen is only part of what we hope will eventually be a comprehensive guide to using TXL in every kind of software analysis and transformation task. We have chosen this set of examples specifically to highlight some of the non-obvious ways in which TXL can be used to efficiently implement many tasks.

By using a range of real problems rather than small toy examples, we have been able to expose a number of paradigms of use that allow TXL to be effective. The real power of the language lies not in its own features, but rather in the way it is used - these solution paradigms. The purpose of the cookbook is to document and demonstrate these paradigms so that potential users can see how to solve their own problems using TXL and similar tools.

References

1. Barnard, D.T., Holt, R.C.: Hierarchic Syntax Error Repair for LR Grammars. Int. J. Computing and Info. Sci. 11(4), 231–258 (1982)
2. Baxter, I., Pidgeon, P., Mehlich, M.: DMS: Program Transformations for Practical Scalable Software Evolution. In: Proc. Int. Conf. on Software Engineering, pp. 625–634. ACM Press, New York (2004)
3. Bergstra, J.A., Heering, J., Klint, P.: Algebraic Specification. ACM Press, New York (1989)
4. Beyer, D.: Relational programming with CrocoPat. In: Proc. Int. Conf. on Software Engineering, pp. 807–810. ACM Press, New York (2006)
5. van den Brand, M., Klint, P., Vinju, J.J.: Term Rewriting with Traversal Functions. ACM Trans. on Software Eng. and Meth. 12(2), 152–190 (2003)
6. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A Language and Toolset for Program Transformation. Sci. Comput. Program. 72(1-2), 52–70 (2008)
7. Cordy, J.R., Dean, T.R., Malton, A.J., Schneider, K.A.: Source Transformation in Software Engineering using the TXL Transformation System. J. Info. and Software Tech. 44(13), 827–837 (2002)
8. Cordy, J.R.: The TXL Source Transformation Language. Sci. Comput. Program. 61(3), 190–210 (2006)

9. Cordy, J.R.: Source Transformation, Analysis and Generation in TXL. In: Proc. ACM SIGPLAN Works. on Partial Eval. and Program Manip., pp. 1–11. ACM Press, New York (2006)
10. Cordy, J.R.: The TXL Programming Language, Version 10.5. Queen's University at Kingston, Canada (2007), http://www.txl.ca/docs/TXL105ProgLang.pdf
11. Cordy, J.R., Visser, E.: Tiny Imperative Language, http://www.program-transformation.org/Sts/TinyImperativeLanguage
12. Cordy, J.R.: The TIL Chairmarks, http://www.program-transformation.org/Sts/TILChairmarks
13. Dean, T.R., Cordy, J.R., Malton, A.J., Schneider, K.A.: Agile Parsing in TXL. J. Automated Softw. Eng. 10(4), 311–336 (2003)
14. van Deursen, A., Kuipers, T.: Building Documentation Generators. In: Proc. 1999 Int. Conf. on Software Maint., pp. 40–49. IEEE Press, Los Alamitos (1999)
15. Guo, X., Cordy, J.R., Dean, T.R.: Unique Renaming of Java Using Source Transformation. In: Proc. IEEE Int. Works. on Source Code Analysis and Manip., pp. 151–160. IEEE Press, Los Alamitos (2003)
16. Holt, R.C.: An introduction to TA: The Tuple-Attribute Language. Technical report, University of Toronto (1997), http://plg.uwaterloo.ca/~holt/papers/ta-intro.htm
17. Holt, R.C.: Structural Manipulations of Software Architecture using Tarski Relational Algebra. In: Proc. Int. Working Conf. on Reverse Eng., pp. 210–219. IEEE Press, Los Alamitos (1998)
18. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Trans. Software Eng. 28(7), 654–670 (2002)
19. Moonen, L.: Generating Robust Parsers using Island Grammars. In: Proc. Int. Working Conf. on Reverse Eng., pp. 13–22. IEEE Press, Los Alamitos (2001)
20. Vinju, J., Klint, P., van der Storm, T.: Rascal: a Domain Specific Language for Source Code Analysis and Manipulation. In: Proc. Int. Working Conf. on Source Code Analysis and Manip., pp. 168–177. IEEE Press, Los Alamitos (2009)
21. Martin, J.: RSF file format. Technical report, University of Victoria (August 1999), http://strategoxt.org/Transform/RigiRSFSpecification
22. Weiser, M.D.: Program slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method. University of Michigan, Ann Arbor (1979)

Model Synchronization: Mappings, Tiles, and Categories

Zinovy Diskin

Generative Software Development Lab.,
University of Waterloo, Canada
zdiskin@gsd.uwaterloo.ca

Abstract. The paper presents a novel algebraic framework for specification and design of model synchronization tools. The basic premise is that synchronization procedures, and hence algebraic operations modeling them, are *diagrammatic*: they take a configuration (diagram) of models and mappings as their input and produce a diagram as the output. Many important synchronization scenarios are based on diagram operations of square shape. Composition of such operations amounts to their *tiling*, and complex synchronizers can thus be assembled by tiling together simple synchronization blocks. This gives rise to a visually suggestive yet precise notation for specifying synchronization procedures and reasoning about them.

1 Introduction

Model driven software engineering puts models at the heart of software development, and makes it heavily dependent on intelligent model management (MMt) frameworks and tools. A common approach to implementing MMt tasks is to present models as collections of objects, and program model operations as operations with these objects; *object-at-a-time* programming is a suitable name for this style [1]. Since models may contain thousands of interrelated objects, object-at-a-time programming can be very laborious and error-prone. In a sense, it is similar to the infamous record-at-a-time programming in data processing, and has similar problems of being too close to implementation.

Replacing record- by relation-at-a-time frameworks has raised data processing technology to a qualitatively new level in semantic transparency and programmers' productivity. Similarly, we can expect that *model-at-a-time* programming, in which an engineer can think of MMt routines in terms of operations over models as integral entities, could significantly facilitate development of MMt applications [1]. This view places MMt into the realm of algebra: models are indivisible points and model manipulation procedures are operations with them.

Model synchronization tools based on special algebraic structures called *lenses* [2] can be seen as a realization of the algebraic vision. The lens framework was first used for implementing a bidirectional transformation language for synchronizing simple tree structures [3], and then employed for building synchronization

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 92–165, 2011.

tools for more complex models closer to software engineering practice [4,5]. In [6], a lens-like algebraic structure was proposed to model semantics of QVT, an industrial standard for model transformation.

Lens-based synchronization is *discrete*: input data for a synchronizer consist of states of the models only, while mappings (deltas) relating models are ignored. More accurately, the synchronizer itself computes mappings based on keys and the structure of the models involved. However, in general a pair of models does not determine a unique mapping between them. To compute the latter, some context-dependent information beyond models may be needed, and hiding model mappings inside the tool rather than allowing the user to control them may compromise synchronization. For example, discrete composition of model transformations may be erroneous because in order to be composable, transformations must fit together on both models *and* mappings. In the paper we will consider several examples showing that model (and metamodel) mappings are crucial for model synchronization, and must be treated as first-class citizens not less important than models.

In algebraic terms, the arguments above mean that model mappings must be explicitly included in the arity shapes of MMt operations. A typical MMt universe should appear as a directed graph (nodes are models and arrows are mappings) that carries a structure of *diagrammatic* algebraic operations. The latter act upon configurations (diagrams) of models and mappings of predefined arity shapes: take a diagram as the input and produce a diagram as the output.

The world of diagram algebra essentially differs from the ordinary algebra. A single diagram operation may produce several nodes and arrows that must satisfy certain incidence relationships between themselves and input elements. Composition of such operations, and parsing of terms composed from them, are much more complex than for ordinary tuple-based single-valued operations. Fortunately, we will see that diagram operations appearing in many model synchronization scenarios have a square shape: the union of their input and output diagrams is a square composed of four arrows — we will call it a *tile*. Composition of such operations amounts to their *tiling*, and complex synchronization scenarios become *tiled*. Correspondingly, complex synchronizers can be assembled by tiling together simpler synchronizing blocks, and their architecture is visualized in a precise and intuitive way.

The main goal of the paper is to show the potential of the tile language for specifying synchronization procedures and for stating the laws they should satisfy. Tiles facilitate thinking and talking about synchronization; they allow us to draw synchronization scenarios on the back of an envelope, and to prove theorems about them as well. Specification and design with tiles are useful and enjoyable; if the reader will share this view upon reading the paper, the goal may be considered achieved.

How to read the paper. There are several ways of navigating through the text. The fastest one is given by the upper lane in Fig. 1: rectangles denote sections (of number n)) and arrows show logical dependencies between them.

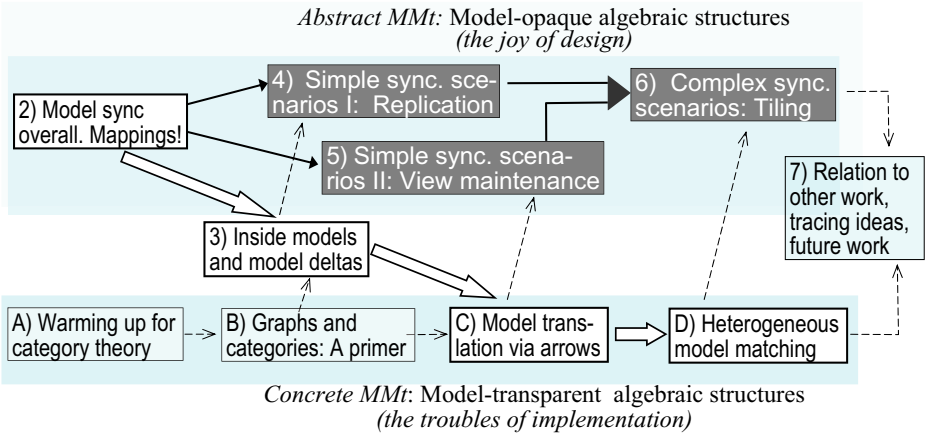


Fig. 1. Flow of Content

Section 2 is the beginning of the journey: it draws an overall picture of model synchronization, presents two simple examples (replica synchronization and view maintenance), and argues that mappings are of primary importance. It also warns the reader about the dangers of walking through *the arrow forest* and declares tile algebra and category theory as a means to meet the challenge.

The subsequent three upper sections present abstract algebraic models of the examples from Section 2, and develop them into an algebraic framework based on tiles. Models and model mappings are treated as opaque indivisible nodes and arrows, and synchronization procedures as abstract algebraic operations over them. Two families of such operations are considered for two basic scenarios: replication (Section 4) and view maintenance (Section 5). Section 6 shows how to build complex synchronizers by putting together basic blocks.

The upper three sections can be viewed as a mini-tutorial on building algebraic theories in the diagrammatic setting. We will see how to set signatures of diagram operations, state equational laws, and define diagram algebras intended to model synchronization tools. The goal is to present a toolbox of algebraic instruments and show how to use them; several exercises should allow the reader to give them a try. Except in subsection 6.2, the mathematics employed in the upper lane is elementary (although somewhat unusual).

The upper lane of the paper presents an *abstract* MMt framework: models and mappings are black-boxes without internal structure (hence its notation: black opaque nodes and arrows). This setting can be useful for a top-level architectural design of synchronization tools. A more refined (and closer to implementation) setting is presented in the *concrete* MMt branch of the paper formed by Sections 3,C,D connected by transparent arrows. In these sections we look inside models and mappings, consider concrete examples, and refine the abstract constructs of the upper lane by more "concrete" algebraic models. In more detail, Section 3 factorizes the fast route $2 \to 4$ (from examples in Section 2 to abstract constructs in Section 4) by providing a formal model for the internal structure of

models and model deltas, and for delta composition as well (including deltas with inconsistencies!). Section C refines the fast route $2 \to 5$ into a "concrete" path $2 \to 3 \to C \to 5$ by providing an algebraic model for the view mechanism (also based on tiles); and Section D plays a similar role for Section 6 with a refined model of heterogenous matching.

Both frameworks — abstract and concrete — employ algebraic models and tiling. A principal distinction of the latter is that metamodels and metamodel mappings are explicitly included into algebraic constructs and play an essential role. Indeed, ignoring metamodels and their mappings hides semantic meaning of operations with heterogeneous models from the user and may provoke ad hoc solutions in building MMt-tools. Taking metamodels seriously brings onto the stage an entire new dimension and significantly complicates the technical side of mapping management. Use of category theory (CT) seems unavoidable, and two "concrete" sections C and D require certain categorical intuition and habits of arrow thinking not familiar to the MMt community.[1] Therefore, a special "starter" on CT was written (Sect. A), which motivates and explains the basics of arrow thinking. Section B is merely a technical primer on graphs and categories: it fixes notation and defines several basic constructs employed in the paper (but is not intended to cover all categorical needs). Even though the presentation in Sect. C and D is semi-formal, all together the four lower sections are much more technically demanding than the upper ones, and so are placed in the Appendix that may be skipped for the first reading.

Sections 7 presents diverse comments on several issues considered or touched on in the paper in a wider context. It also briefly summarizes contributions of the paper and their relation to other work. Section 8 concludes. Answers to exercises marked by * can be found on p. 143

A possible reading scenario the author has in mind is as follows. The reader is a practitioner with a solid knowledge of model synchronization, who knows everything presented in the paper but empirically and intuitively. He has rather vague (if any) ideas about diagram algebra and category theory, and is hardly interested in these subjects, yet he may be interested in a precise notation for communicating his empirical knowledge to his colleagues or/and students. He may also be interested in some mathematics that facilitates reasoning about complex synchronization procedures or even allows their mechanical checking. Such a reader would take a look through numerous diagrams in the paper with an approximate understanding of what they are talking about, and hopefully could find a certain parallelism between these diagrams and his practical intuition. Perhaps, he would remember some terms and concepts and, perhaps, would take a closer look at those concepts later on. Eventually, he may end up with a feeling that viewing model synchronization through the patterns of diagram algebra makes sense, and category theory is not so hopelessly abstract.

Now it is the reader's turn to see if this scenario is sensible.

[1] It could explain why many known algebraic approaches to MMt ignore the metamodeling dimension.

2 Model Sync: A Tangled Story

By the very nature of modeling, a system to be modeled is represented by a set of interrelated models, each one capturing a specific view or aspect of the system. Different views require different modeling means (languages, tools, and intuitions), and their models are often built by different teams that possess the necessary experience and background. This makes modeling of complex systems heterogeneous, collaborative, and essentially dependent on model synchronization.

This section presents *a tale* of model synchronization: we begin with a tangle, then follow it and get to an arrow forest, which we will try to escape by paving our way by tiles.

2.1 The Tangle of Relationships and Update Propagation

The task of model synchronization is schematically presented in Fig. 2. A snapshot of a design project appears as a heterogeneous collection $\mathcal{M}$ of models (shown by nodes $A, B, C...$) interrelated in different ways (edges $r_1, r_2, r_3...$). The diversity of node and edge shapes is a reflection of the diversity of models and the complexity of their mutual relationships that emerge in software design. The image of a tangle in the center of the figure is intentional.

Typically, models in a project's snapshot are only partially consistent, i.e., their relationships partially satisfy some predefined consistency conditions. That is, we suppose that inconsistencies are partially detected, specified and recorded for future resolution. Inconsistency specifications may be considered as part of the intermodel relationships and hence are incorporated into intermodel edges.

Now suppose that one of the models (say, A in the figure) is updated to a new state (we draw an arrow $u_A: A \to A'$), which may violate existing consistent relationships and worsen existing inconsistencies. To restore consistency or at least to reduce inconsistency, other related models must be updated as well (arrows $u_B: B \to B'$, $u_C: C \to C'$ *etc*). Moreover, relationships between models must also be updated to new states r'_i, $i = 1, 2, ...$, particularly by incorporating new inconsistencies. Thus, the initial update u_A is to be *propagated* from the updated model to other related models and relationships so that the entire related fragment ("section" $\mathcal{M}$ of the model space) is updated to state $\mathcal{M}'$. We call this scenario a *single-source update propagation.*

Another scenario is when several models (say, A, B, C) are updated concurrently, so their updates must be *mutually* propagated between themselves and other models and relationships. Such *multi-source* propagation is more complex because of possible conflicts between updates. However, even for single-source propagation, different propagation paths may lead to the same model and generate conflicts; cycles in the relationship graph confuse the situation even more. The relationship tangle generates a propagation tangle.

Propagation is much simpler in the binary case when only two interrelated models are considered. This is a favorite case of theoreticians. For binary situations, multi-source propagation degenerates into *bi-directional* (in contrast to *unidirectional* single-source propagation) — an essential simplification but still

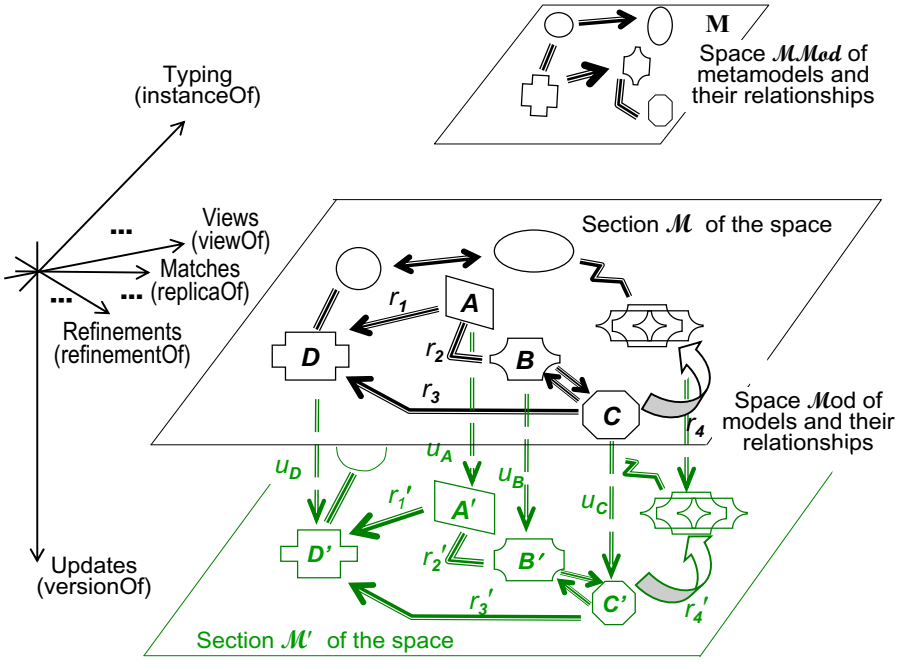


Fig. 2. Models and their relationships: From a tangle to mD-space

a challenge [7]. Practical situations enjoy a mix of single- and multi-source, uni- and bi-directional propagations. We will generically refer to them as *synchronization procedures*.

The description above shows that understanding intermodel relationships is crucial for design of synchronization procedures, and it makes sense to establish a simple taxonomy. For the binary case, one model in relation to another model may be considered as its

- replica (e.g., a Google replica of a Microsoft Outlook calendar),
- updated version (two versions of the same replica),
- view (a business view of a calendar),
- refinement (an hourly refinement of a daily schedule),
- instance (an actual content of a diary book – the metamodel for the content).

The list could be be extended and gives rise to a family of binary relations $\mathcal{R}_i \subset \mathcal{M}od \times \mathcal{M}od$, $i = 1, 2..$ over the space of models $\mathcal{M}od$. Unfortunately, a more or less complete classification of such relations important for MMt seems to be missing from the literature.

An observation of fundamental importance for model synchronization is that intermodel relationships are not just pairs of models $(A, B) \in \mathcal{R}_i$, they are mappings $r\colon A \Rightarrow B$ linking models' elements. That is, edges in Fig. 2 have extension consisting of links. Roughly, we may think of an edge $r\colon A \Rightarrow B$ as a set of ordered pairs (a, b) with $a \in A$ and $b \in B$ being similar model elements

(a class and a class, an attribute and an attribute *etc*). We may write such a pair $\ell{=}(a,b)$ as an arrow $a \xrightarrow{\ell:r} b$ and call it a *link* (note the difference in the bodies of arrows for mappings and links). In the UML jargon, links ℓ are called *instances* of r. In the arrow notation for links as above, the name of the very link ℓ may be omitted but the pointer to its type, $:r$, is important and should be there.

Table 1. Intermodel relationships & mappings

Relationship	Mapping
replicaOf	match
versionOf	update
viewOf	view trc.
instanceOf	typing

Table 1 presents a brief nomenclature of intermodel relations and mappings ('trc.' abbreviates 'traceability'). Normally mappings have some structure over the set of links they consist of, and we should distinguish between a mapping r and its extension $|r|$, i.e., the set links the mapping consists of. Yet we will follow a common practice and write $\ell \in r$ for $\ell \in |r|$. In general, a mapping's extra structure depends on the type of the relationship, and so mappings listed in the table are structured differently and operated differently.

2.2 Mappings, Mappings, Mappings...

In this section we consider how mappings work for synchronization. We will begin with two simple examples. The first considers synchronization of two replicas of a model. In the second, one model is a view of the other rather than an independent replica. Then we will discuss deficiencies of state-based synchronization. Finally, we discuss mathematics for mapping management.

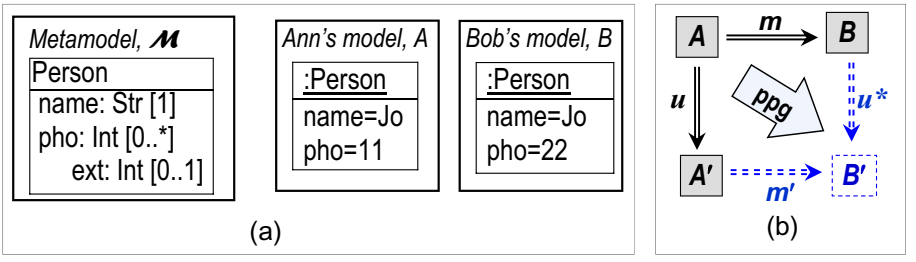
To make tracing examples easier, our sample models will be object diagrams, whose class diagrams thus play the role of metamodels (and the metamodel of class diagram is the meta-metamodel).

2.2.1 Replica Synchronization

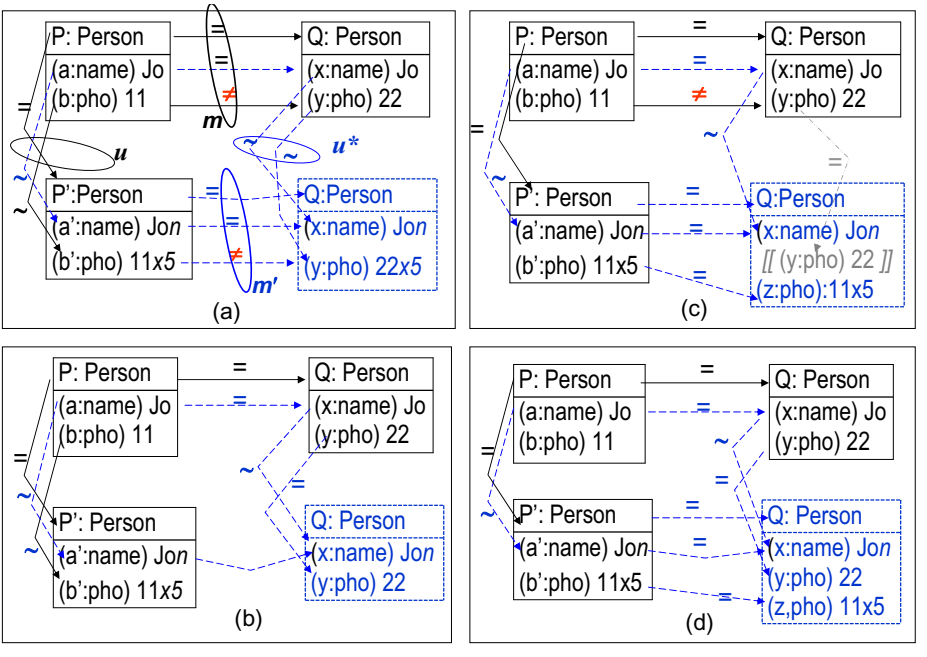
Suppose that two developers, Ann and Bob, maintain their own replicas of a simple model Fig. 3i(a). The model consists of Person-objects with mandatory attribute 'name' and any number of 'phone's with an optional extension number 'ext' (see the metamodel in the leftmost square; attribute multiplicities are shown in square brackets).

Diagram in Fig. 3i(b) presents an abstract schema of a simple synchronization scenario. Arrow $m\colon A \Rightarrow B$ denotes some correspondence specification, or a *match*, between the models. Such specifications are often called *(symmetric) deltas*, and are computed by model differencing tools.[2] Similarly, arrow $u\colon A \Rightarrow A'$ denotes the delta between two versions of Ann's replica, and we call it an *update*. The task is to propagate this update to Bob's replica and update the match. That is, the propagation operation `ppg` must compute an updated model B' together with update u^* and new match m'. Note that derived arrows are dashed (and the derived node is blank, rather than shaded). When reading

[2] The term *directed delta* refers to an operational (rather than structural) specification: a sequence of operations (add, change, delete) transforming A to B (an edit log).



i) Two simple replicas to be synchronized



ii) Four cases with different input mappings

Fig. 3. Mappings do matter in update propagation

the paper in color, derived elements would be blue (because the color blue reminds us of machines and mechanical computation). We will continue with this pattern throughout the paper.

Fig. 3ii demonstrates that the results of update propagation depend on the input mappings u and m. All four cases presented in the figure have the same input models A, A', B, but different mappings m or/and u, which imply — as we will see — different outputs B', u^*, m'.

Consider Fig. 3ii(a). Models' elements (their OIDs) are denoted by letters $P, a, ..., Q, x, ...$ We match models by linking those elements that are different replicas of the same objects in the real world (note the label =). Some of such links are provided by the user (or a matching tool) while others can be derived using the metamodel. For example, as soon as elements $P@A$ and $Q@B$ are linked, their 'name' attributes must be linked too because the metamodel prescribes a mandatory unique name for any Person object. In contrast, linking the phone attributes $b@A$ and $y@B$ is an independent (basic rather then derived) datum because the metamodel allows a person to have several phones. The match shown in the figure says that b and y refer to the same phone. Then we have a conflict between models because they assign different numbers to the same phone.In such cases the link is labeled by (red) symbol $\neq$ signaling a conflict. The set of all matching links together with their labels is called a *matching mapping* or just a *match*, $m\colon A \Rightarrow B$.

An *update mapping* $u\colon A \Rightarrow A'$ specifies a delta between models in time. Mapping u in Fig. 3ii(a) consists of three links. Note that in general the OIDs of the linked (i.e., the "same") objects may be different if, for example, Ann first deleted object P but later recognized that it was a mistake and restored it from scratch as a fresh object P'. Then we must explicitly declare the "sameness" of P and P', which implies the sameness of their 'name' attributes. In contrast, the sameness of phone numbers is an independent datum that must be explicitly declared. Different values of linked attributes mean that the attribute was modified, and such links are labeled by $\sim$ (the update analog of $\neq$-label for matches).

Now we will consecutively consider the four cases of update propagation shown in Fig. 3ii. In all four cases, link $PP' \in u$ means that object P is not deleted, and hence its model B's counterpart, object Q, is also preserved (yet in Fig. 3ii =-links QQ in mapping u^* are skipped to avoid clutter.) However, Q's attribute values are kept unchange or modified according to mappings u and m.

Case (a). Name change in A is directly propagated to B, and addition of phone extension specified by u is directly propagated to u^*. The very phone number is not changed because match m declared a conflict, and our propagation policy takes this into account. A less intelligent yet possible policy would not propagate the extension and keep the entire y unchanged.

Case (b): conflicting link $b \to y$ is removed from the match, i.e., Ann and Bob consider different phones of Jo. Hence, the value of $y@B$ should not change.

Case (c): link $b \to b'$ is removed from mapping u, i.e., Jo's phone b was deleted from the model, and a new phone b' is added. Propagation of this update can be managed in different ways. For example, we may require that deletions are

propagated over both =- or ≠-matching links, and then phone y must be deleted from B'. Or we may set a more cautious policy and do not propagate deletions over conflicting matching links. Then phone y should be kept in B' (this variant is shown in square brackets and is grey). Assuming that additions to A are always propagated to B, we must insert in B a new phone z "equal" to b'.

Case (d) is a superposition of cases (b,c): both links $b \to y$ and $b \to b'$ are removed from resp. m and u. A reasonable update policy should give us model B' as shown: phone y is kept because it was not matched to the deleted b, and phone z is the new b' propagated to B'. This result can be seen as a superposition of the results in (b) and (c), and our propagation policies thus reveal compatibility with mappings' superposition.

Discussion. In each of the four cases we have an instance of the operation specified in Fig. 3i(b): given an input diagram (u, m), an output diagram (u^*, m') is derived. What we call an update propagation policy is a precise specification of how to build the output for any input. Three points are worth mentioning.

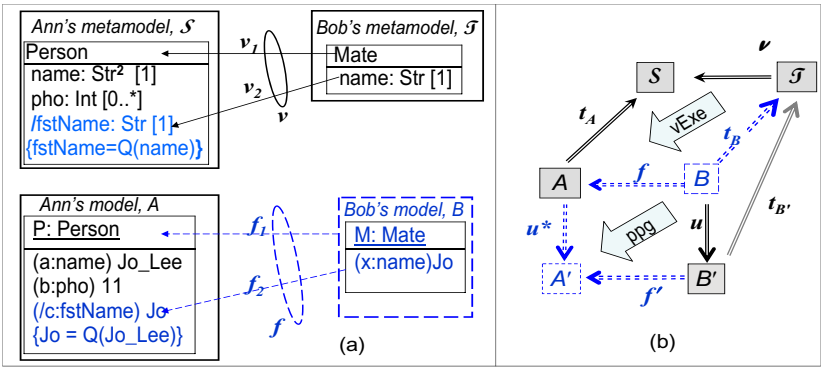
1. Policies are based on the metamodel: for example, a policy may prescribe different propagation strategies for different attributes (say, phone changes are propagated but name changes are not).
2. Recall that in cases (a,c) we discussed different possibilities of update propagation. They correspond to different policies rather than to different outputs of a single policy. That is, different policies give rise to different algebraic operations but a given policy corresponds to a deterministic operation producing a unique output for an input.
3. The mapping-free projection of the four cases would reveal a strange result: the same three input models A, B, A' generate different models B' for a given policy. That is, the mapping-free projection of a reasonable propagation procedure cannot be seen as an algebraic operation.

2.2.2 View Update Propagation

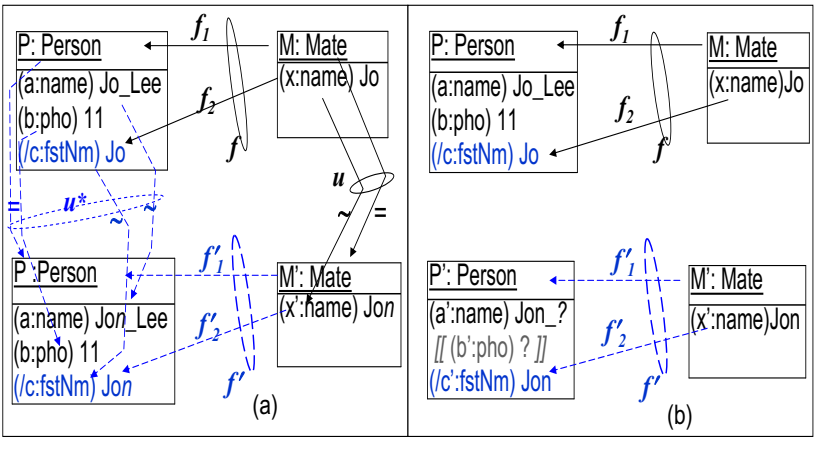
Now we consider a different situation when Bob's model is a view of Ann's one, see Fig. 4i(a). Ann is interested in objects called Persons, their full names, i.e., pairs (fstName, lstName), and phone numbers. Bob calls these objects Mates, and only considers their first names but call the attribute 'name'.

To specify this view formally, we first augment Ann's metamodel $\boldsymbol{\mathcal{S}}$ with a derived attribute 'fstName' coupled with the query specification Q defining this element. Query Q says "take the first component of a name"; formally, fstName$\stackrel{\text{def}}{=} Q$(name)=$\mathsf{proj}_1$(name). Then we map Bob's metamodel $\boldsymbol{\mathcal{T}}$ into Ann's one as shown in the figure, where the view definition mapping $\boldsymbol{v}: \boldsymbol{\mathcal{T}} \Rightarrow \boldsymbol{\mathcal{S}}$ consists of two links. Link v_1 says that Bob's class Mate is Ann's class Person. Link v_2 says that Mate attribute 'name' is Person's 'fstName' computed by query Q.

Now let A be a model over Ann's metamodel $\boldsymbol{\mathcal{S}}$ shown in the left lower corner of Fig. 4i(a). We may apply to it the query Q specified in the metamodel, and compute the derived attribute $c = \mathsf{proj}_1$(Jo_Lee)=Jo. Then we select those elements of the model, whose types are within the range of mapping $\boldsymbol{v}$, and relabel them according to this mapping.



i) Propagating view update to the source



ii) Two cases with different update mappings

Fig. 4. Mappings do matter in update propagation cont'd

The result is shown in the right-lower corner as model B, and links $f_{1,2}$ trace the origin of its elements. These links constitute the traceability mapping $f = \{f_1, f_2\}$. In this way, having the view definition mapping $\boldsymbol{v}$, any Ann's model A (an instance of $\boldsymbol{S}$) can be translated into a $\boldsymbol{T}$'s instance B computed together with traceability mapping $f\colon B \Rightarrow A$. (A more complex example can be found in Sect. C.)

Thus, we have a diagram operation specified by square diagram $A\boldsymbol{ST}B$ in Fig. 4i(b). It takes two mappings — view definition $\boldsymbol{v}$ and typing of the source model t_A, and produces model B (together with its typing t_B) and traceability mapping $f\colon B \Rightarrow A$. This is nothing but an arrow formulation of the view execution mechanism; hence the name vExe of the operation.

Now suppose that the view is updated with mapping $u\colon B \Rightarrow B'$, and we need to propagate the update back to the source as shown by the lower square in Fig. 4i(b). Update propagation is a different type of diagram operation, and it is convenient to consider the two diagrams as orthogonal: view execution is the top face of the semi-cube and propagation is the front. Note that an output element of operation vExe, mapping f, is an input element for operation ppg; diagram Fig. 4i(b) thus specifies substitution of one term into another (and we have an instance of tiling mentioned above).

Fig. 4ii presents two cases of update propagation. In case (a), the name of Mate-object M was modified, and this change is propagated to object P – the preimage of M in the source model. Elements of model A not occurring in the view are kept unchanged. In case (b), the update mapping is empty, which means that object M was deleted and a new object M' added to the model. Correspondingly, object P is also deleted and a new object P' is added to A. Since the view ignores last names and phones numbers, these attributes of P' are set to Unknown (denoted by ?). The attribute b' is shown in brackets (and grey) because a different propagation policy could simply skip P''s phone number as it is allowed by the metamodel (but the last name cannot be skipped and its value must be set to Unknown).

The results of Discussion at the end of the previous section applies to the view update propagation as well.

2.2.3 Why State-Based Synchronization Does Not Work Well

Examples above show that synchronization is based on mappings providing model alignment, particularly, update mappings. Nevertheless, *state-based* frameworks are very popular in data and model synchronization. Being state-based means that the input and the output of the synchronizer only include states of the models while update mappings are ignored. More accurately, model alignment is done inside the synchronizer, as a rule, on the basis of keys (names, identifying numbers or other relevant information, e.g., positions inside a predefined structure). However, this setting brings with it several serious problems.

First of all, update mappings cannot be, in general, derived from the states. Identification based on names fails in cases of synonymy or homonymy that are not infrequent in modeling. Identification numbers may also fail, e.g., if an employee quit and then was hired back, she may be assigned a new identification

number. “Absolutely” reliable identification systems like SSNs are rarely available in practice, and even if they are, fixing a typo in a SSN creates synonymy. On the other hand, identification based on internal immutable OIDs also does not solve the problem if the models to be aligned reside in different computers. Even for models in the same computer, OID-based identification fails if an object was deleted but then restored from scratch with a new ID, not to mention the technological difficulties of OID-based alignment. Thus, update mappings cannot be computed entirely automatically, and there are many model differencing tools [8,9,10] employing various heuristics and requiring user assistance to fix the deficiencies of the automatic identification. In general, alignment is another story, and it is useful to separate concerns: discovering updates and propagating updates are two different tasks that must be treated differently and addressed separately.

Second, writing synchronization procedures is difficult and it makes sense to divide the task into simpler parts. For example, view update propagation over a complex view can be divided into composition of update propagations over the components as shown in Fig. 5: $\boldsymbol{\mathcal{X}}$ is some intermediate metamodel and view definition $\boldsymbol{v}$ is composed from parts, $\boldsymbol{v} = \boldsymbol{v}_1; \boldsymbol{v}_2$. It is reasonable to compose the procedure of update propagation over view $\boldsymbol{v}$ from propagation procedures over the component views as shown in the figure. It is a key idea for the lens approach to tree-based data synchronization [2], but lens synchronization is state-based and so two propagation procedures $\mathtt{ppg}_1$ and $\mathtt{ppg}_2$ can be composed if the output states of the first are the input for the second. Hence, the composed procedure will be erroneous if the components use different alignment strategies (e.g., based on different keys) and then we have different update mappings u^1_X, u^2_X as shown in the figure.

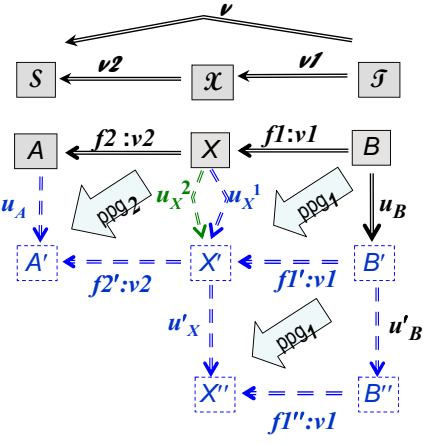


Fig. 5. Mappings do matter in update propagation (cont'd)

Finally, propagation procedures are often compatible with update composition: the result of propagating a composed update $u_B; u'_B$ is equal to composition of updates $u_X; u'_X$ obtained by consecutive application of the procedure. However, if alignment is included into propagation, this law rarely holds — see [11] for a detailed discussion.

2.3 The Arrow Forest and Categories

Mappings are two-folded constructs. On one hand, they consist of directed links and can be sequentially composed; the arrow notation is very suggestive in this respect. On the other hand, mappings are *sets* of links and hence enjoy set

operations (union, intersection, difference) and the inclusion relation (defined for mappings having the same source and target). Mappings can also be composed in parallel: given $m_i : A_i \Rightarrow B_i$ $(i = 1, 2)$, we can build $m_1 \otimes m_2 : A_1 \otimes A_2 \Rightarrow B_1 \otimes B_2$, where $\otimes$ may stand for Cartesian product or disjoint union (so that we have two types of parallel composition).

Mapping compositions complicate the *relationship tangle* in Fig. 2 even more: the set of basic relationships generates derived relationships. If the latter are not recognized, models remain unsynchronized and perhaps inconsistent. Living with inconsistencies [12] is possible if they are explicit and specified; implicit inconsistencies undermine modeling activities and their automation.

Thus, our tale of unraveling the tangle of relationships led us to an *arrow forest*. Updates, matches, traceability and typing mappings are all important for model synchronization. Together they give rise to complex structures whose intelligent mathematical processing is not evident and not straightforward.

In the paper we will only consider one side of the rich mapping structure: directionality and sequential composition. Even in this simplified setting, specifying systems of heterogeneous mappings needs special linguistic means: right concepts and a convenient notation based on them. Fortunately, such means were developed in category theory and are applicable to our needs (the reader may think of "paved trails in the arrow forest"); the concrete MMt sections of the paper will show how they work.

Arrows of different types interact in synchronization scenarios and are combined into tiles. The latter may be either similar and work in the same plane, or be "orthogonal" and work in orthogonal planes as, for example, shown in Fig. 4i(b). Complex synchronization scenarios are often multi-dimensional and involve combinations of low-dimensional tiles into higher-dimensional ones. For example, update propagation for the case of two heterogeneous models with evolving metamodels gives rise to a synchronization cube built from six 2D-tiles (Sect. 6.2). Higher-dimensional tiles are themselves composable and also form category-like structures. In this way the tangled collection of models and model mappings can be unraveled into a regular net in a multi-dimensional space, as suggested by the frame of reference on the left of Fig. 2. (Note that we do not assume any metric and the space thus has an algebraic rather than a geometric structure. Nevertheless, multi-dimensional visualization is helpful and provides a convenient notation.)

3 Inside Models and Model Deltas

Diagrammatic models employ a compact *concrete* syntax, which is a cornerstone of practical applications. This syntax hides a rich structure of relationships and dependencies between model's elements (*abstract* syntax), which does matter in model semantics, and in establishing relations between models as well. In this section we will take a look "under the hood" and consider structures underlying models (Sect. 3.1) and symmetric deltas (binary relations) between models (Sect. 3.3). To formalize inconsistencies, we introduce *object-slot-value* models

and their mappings (Sect. 3.2). We will use the notions of graph, graph mapping (morphism) and span; their precise definitions can be found in Appendix B.

3.1 Inside Models: Basics of Meta(Meta)Modeling

A typical format for internal (repository) model representation is, roughly, a containment tree with cross-references, in fact, a directed graph. The elements of this graph have attributes and types; the latter are specified in the metamodel. An important observation is that assigning types to model elements constitutes a mapping $t\colon A \to M$ between two graphs underlying the model (A) and its metamodel (M) resp. What is usually called a *model graph* [9,10,13] is actually an encoding of a typing mapping t. Making this mapping explicit is semantically important, especially for managing heterogeneous model mappings.

Example. The upper half of Fig. 6 presents a simple metamodel $\boldsymbol{\mathcal{A}}$ (in the middle) and its simple instance, model A (on the left), with a familiar syntax of class and object diagrams. The metamodel is a class diagram declaring class Person with two attributes. Expressions in square brackets are *multiplicities*: their semantic meaning is that objects of class Person have one and only one name (multiplicity [1..1] or [1] in short), and may have any number of phones, perhaps none (multiplicity [0..*]).

Symbols in round brackets are beyond UML and say whether or not the value of the attribute may be set to Unknown (*null*, in the database jargon). Marking an attribute by ? means that nulls are allowed: every person has a name but it may be unknown; we call attributes *uncertain*. An attribute is called *certain* (and marked by !) if nulls are not allowed and the attribute must always have an actual value. If a person has a phone, its number cannot be skipped.

Model A is an object instance of $\boldsymbol{\mathcal{A}}$. It declares two Person objects: one with an unknown name (which is allowed by the metamodel) and phone number 11, and the other with name Jo and without phones (which is also allowed). Symbol '?' is thus used as both a quasi-value (null) in the models and a Boolean value $? \in \{?, !\}$ in the metamodel.

In its turn, the metamodel is an instance of the meta-metamodel specified by a class diagram $\boldsymbol{\mathcal{M}}$ in the right upper corner. It says that metamodels can declare classes that own any number (perhaps, zero) of attributes, but each attribute belongs to one and only one class (this is a part of the standard semantics for "black diamond" asscoiations in UML). Each attribute is assigned one primitive type, a pair of integers specifying its multiplicity, and a Boolean value for certainty; neither of these can be skipped (marker !). We will use model element names (like Person, pho, etc) as OIds, and hence skip the (important) part of $\boldsymbol{\mathcal{M}}$ specifying element naming: certainty and uniqueness of names.

Remark 1. As is clear from the above, an attribute's multiplicity and certainty are orthogonal concepts. Below we will see that their distinction matters for model synchronization. It also matters for query processing and is well known in the database literature [14]. Surprisingly, the issue is not recognized in UML,

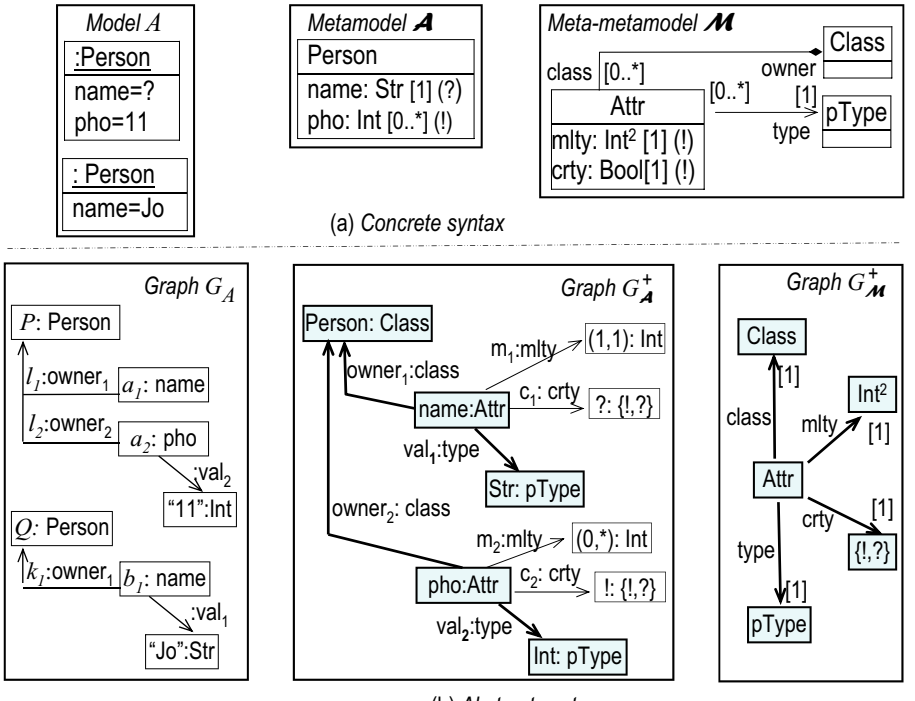


Fig. 6. From models to graphs

whose metamodel for class diagrams does not have the concept of certainty, and handbooks suggest modeling an attribute's uncertainty by multiplicity [0..1] [15].

Example cont'd: Abstract syntax. In the lower half of Fig. 6, the concrete syntax of model diagrams is unfolded into directed graphs: model elements are nodes and their relationships are arrows. We begin our analysis with the metamodel graph $G^+_{\mathcal{A}}$ (in the middle of the figure). Bold shaded nodes stand for the concepts (types) declared in the class diagram $\mathcal{A}$: class Person and its two attributes. Bold arrows relate attributes with their owning class and value domains. The bold elements together form an *instantiable* subgraph $G_{\mathcal{A}}$ of the entire graph $G^+_{\mathcal{A}}$. Non-instantiable elements of $G^+_{\mathcal{A}}$ specify constraints on the intended instantiations.

Graph G_A (the leftmost) corresponds to the object diagram A and specifies an instantiation of graph $G_{\mathcal{A}}$. Each G_A's element has a type (referred to after the colon) taken from graph $G_{\mathcal{A}}$. Nodes typed by Person are *objects* (of class Person) and nodes typed by attributes are *slots* (we use a UML term). Slots are linked to their owning objects and to values they hold. Slot a_1 is *empty*: there are no value links going from it. Thus, the abstract syntax structure underlying a class diagram is a graph $G^+_{\mathcal{A}}$ containing an instantiable subgraph $G_{\mathcal{A}}$ and noninstantiable constraints. A legal instance of graph $G_{\mathcal{A}}$ is a graph mapping $t_A : G_A \to G_{\mathcal{A}}$ satisfying all constraints from $G^+_{\mathcal{A}} \setminus G_{\mathcal{A}}$.

The same pattern applies to the pair $(G^+_{\mathcal{A}}, G_{\mathcal{M}})$, where $G_{\mathcal{M}}$ is the instantiable subgraph of graph $G^+_{\mathcal{M}}$ specifying the metametamodel (the rightmost in Fig. 6). Multiplicities in Fig. 6(b) are given in the sugared syntax with square brackets, and can be converted into nodes and arrows as it is done for graph $G^+_{\mathcal{A}}$; association ends without multiplicities are assumed to be [0..*] by default. Finally, there is a metameta... graph $G_{\mathcal{MM}}$ providing types and constraints for $G^+_{\mathcal{M}}$; it is not shown in the figure.

The entire configuration appears as a chain of graphs and graph mappings in Fig. 7. Horizontal and slanted arrows are typing mappings; vertical arrows are inclusions and symbols $\models$ remind us that typing mappings on the left-above of them must satisfy the constraints specified in the noninstantiable part. This compact specification is quite general and applicable far beyond our simple example. To make it formal, we need to formalize the notion of constraint and its satisfiability by a typing mappings. This can be done along the lines described in [16].

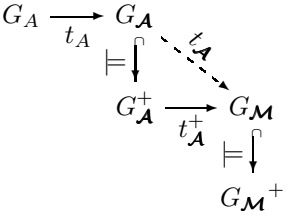


Fig. 7. Models as graphs

Two models are called *similar* if they have the same metamodel, and hence all layers below the upper one are fixed. In our example, two object diagrams are similar if they are instances of the same class diagram.

3.2 Object-Slot-Value Models and Their Mappings

Our definition of models as chains of graph mappings does not distinguish between objects and values: they are just nodes in instance graphs. However, objects and values play different roles in model matching and updating, and for our further work we need to make their distinction explicit. Below we introduce *object-slot-value (osv)* models, whose mappings (morphisms) treat objects and values differently. This is a standard categorical practice: a distinction between objects is explicated via mappings (in Lawvere's words, "to *objectify* means *to mappify*").

In the previous section we defined a metamodel as a graph mapping $t^+_{\mathcal{A}}: G^+_{\mathcal{A}} \to G_{\mathcal{M}}$. Equivalently, we may work with the inverse mapping $(t^+_{\mathcal{A}})^{-1}$, which assigns to each element $E \in G_{\mathcal{M}}$ the set of those $G^+_{\mathcal{A}}$'s elements e for which $t^+_{\mathcal{A}}(e) = E$. It is easy to check that this mapping is compatible with incidence relationships between nodes and arrows and hence can be presented as a graph morphism $(t^+_{\mathcal{A}})^{-1}: G_{\mathcal{M}} \to \boldsymbol{Sets}$ into the universe of all sets and (total) functions between them. (Indeed, multiplicities in graph $G^+_{\mathcal{M}}$ require all its arrows to be functions). To simplify notation, below we will skip the metametamodel's syntax and write E instead of $(t^+_{\mathcal{A}})^{-1}(\mathsf{E})$ (where E stands for Class, Attr, type etc. elements in graph $G_{\mathcal{M}}$). Given a model, we will also consider sets Obj and Slot of all its objects and slots.

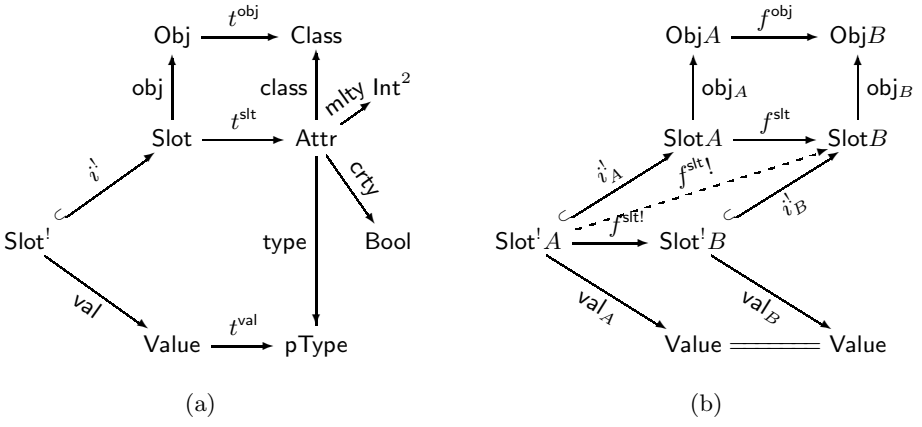


Fig. 8. Osv-models and their mappings

Definition 1. (Osv-models) An *object-slot-value model* is given by a collection of sets and functions (i.e., total single-valued mappings) specified by diagram Fig. 8(a); the hooked arrow $i^!$ denotes an inclusion. The functions are required to make the diagram commutative, and to satisfy two additional constraints (1,2) (related to mlty and crty) specified below after we discuss the intended interpretation of sets and functions in the diagram.

The bottom row gives a system of primitive types for the model, and the right "column" specifies a class diagram without associations (the metamodel). For example, model A in Fig. 6 is an instance of the osv-model definition with sets Class={Person}, Attr={name, pho}, pType={Str, Int} and Value consisting of all strings and all integers. Classes Int and Bool have their usual extension consisting, resp., of integers (including "infinity" *) and Boolean values (denoted by ?,!).[3] The functions are defined as follows: type(name) = Str, type(pho) = Int; class(name) = class(pho) = Person; mlty(name) = (1,1), mlty(pho) = (0,*); crty(name) = !, crty(pho)= ?

The left column specifies the "changeable/run-time" part of the model — an object diagram over the class diagram; hence, there are typing mappings t^{obj}, t^{slt} and the requirement for the upper square diagram to be commutative. For example, for model A in Fig. 6, we have sets Obj = $\{P, Q\}$, Slot = $\{a_1, a_2, b_1\}$ and functions: $t^{obj}(P) = t^{obj}(Q)$ = Person; $t^{slt}(a_1)$ = name, *etc*; obj$(a_1) = P$, *etc.*

Slots in set Slot! are supposed to hold a real value extracted by function val. This value should be of the type specified for the attribute, and the lower polygon is also required to be commutative. Slots in set Slot$^? \stackrel{\text{def}}{=}$ Slot $\setminus$ Slot! are considered empty, and function val is not defined on them. For model A, we

[3] For a punctilious reader, values in classes Int and Bool live in the metalanguage and are different from elements of set Value.

have $\mathsf{Slot}^! = \{a_2, b_1\}$ $\mathsf{val}(a_2) =$ '11', $\mathsf{val}(b_1) =$ 'Jo' whereas $a_1 \in \mathsf{Slot}^?$. We will continue to use our sugared notation $\mathsf{val}(s) =?$ for saying that slot $s \in \mathsf{Slot}^?$ and hence $\mathsf{val}(s)$ is not defined.

The following two conditions hold.
(1) For any attribute $a \in \mathsf{Attr}$ and object o with $t^{\mathsf{obj}}(o) = \mathsf{class}(a)$, if $\mathsf{mlty}(a) = (m, n)$, then $m \le |\mathsf{obj}^{-1}(o)| \le n$ (i.e., the number of a-slots that a $\mathsf{class}(a)$-object has must satisfy a's multiplicity).
(2) If for a slot $s \in \mathsf{Slot}$ we have $s.t^{\mathsf{slt}}.\mathsf{crty} = 1$ (i.e., the attribute is certain), then $s \in \mathsf{Slot}^!$.

Definition 2. (Osv-model mappings) Let A, B be two osv-models over the same class diagram, i.e., they have the same right "column" in diagram Fig. 8(a) but different changeable parts distinguished by indexes A, B added to the names of sets and functions (see Fig. 8(b) where the class diagram part is not shown, and bottom double-line denotes identity). We call such models *similar*.

A *mapping* $f\colon A \to B$ of similar osv-models is a pair $f = (f^{\mathsf{obj}}, f^{\mathsf{slt}})$ of functions shown in Fig. 8(b) such that the upper square in the diagram commutes, and triangles formed by these functions and typing mappings (going into the "depth" of the figure) are also commutative: $f^{\mathsf{obj}}; t_B^{\mathsf{obj}} = t_A^{\mathsf{obj}}$ and $f^{\mathsf{slt}}; t_B^{\mathsf{slt}} = t_A^{\mathsf{slt}}$.

In addition, the following two conditions hold.
(3) Let $f^{\mathsf{slt}}!\colon \mathsf{Slot}^!A \to \mathsf{Slot}B$ be the composition $i^!_A; f^{\mathsf{slt}}$, i.e., the restriction of function f^{slt} to subset $\mathsf{Slot}^!A$. We require function $f^{\mathsf{slt}}!$ to map a non-empty slot to a non-empty slot. Then we actually have a total function $f^{\mathsf{slt}!}\colon \mathsf{Slot}^!A \to \mathsf{Slot}^!B$, and the upper diamond in diagram (b) is commutative.
(4) The lower diamond is required to be commutative as well: a non-empty slot with value x is mapped to a non-empty slot holding the same value x.

To simplify notation, all three components of mapping f will often be denoted by the same symbol f without superscripts.

Remark 2. Condition (3) says nothing about B-slot $f^{\mathsf{slt}}(s)$ for an empty A-slot $s \in \mathsf{Slot}^?A$: it may be be also empty, or hold a real value. That is, a slot with '?' can be mapped to a slot with either '?' or a real value (but a slot with a real value v is mapped to a slot holding the same v by condition (4)).

Commutativity of diagram Fig. 8(b) is the key point of Definition 2 and essentially ease working with model mappings. (Categorically, commutativity means that model mappings are *natural transformations*). This advantage comes for a price: condition (4) prohibits change of attribute values in models related by a mapping, and hence we need to model attribute changes somehow differently. We will solve this problem in the next section.

3.3 Model Matching via Spans

Comparing two models to discover their differences and similarities is an important MMt task called *model differencing* or *matching*. Since absolutely reliable keys for models' elements are rarely possible in practice, model matching tools

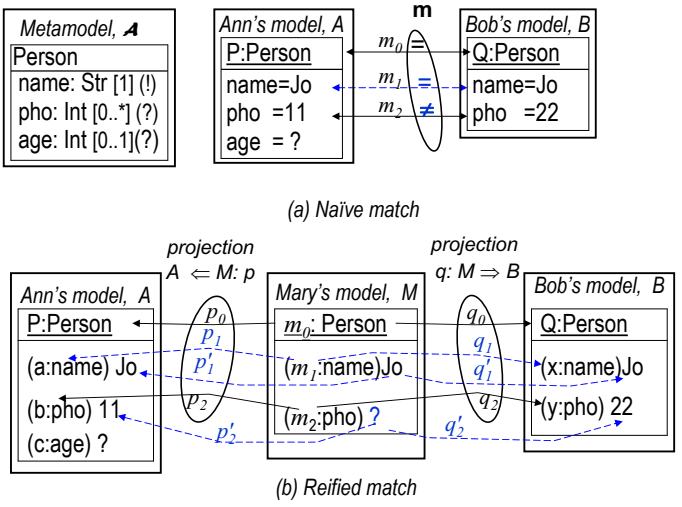


Fig. 9. Reification of matches

usually employ complex heuristics and AI-based techniques (like, e.g., similarity flooding [17]), which are tailored to specific kinds of models or/and to specific contexts of model comparison [8,10,18]. Whatever the technic is used for model matching, the result is basically a set of matching links between the models' elements. Such sets have a certain structure, and our goal in this section is to specify it formally.

A simple example of model matching is shown in Fig. 9(a). Two similar models are matched by a family **m** of links $m_{0,1,2}$ between model elements (objects and slots). Linking slots implies linking their values; hence we have two additional links $m'_1 : \text{Jo} \to \text{Jo}$ and $m'_2 : 11 \to 22$. The latter link shows a *conflict* between the models.

All matching links respect typing: we cannot match an attribute and an object, or two attributes belonging to unmatched classes. The set of matching links is itself structured similarly to models being matched, and hence can be seen as a new model, say, M as shown in Fig. 9(b). (Name M stands for Mary — an MMt administrator who did the comparison of Ann's and Bob's models.) In the UML jargon, this step can be called *reification* of links: each one becomes an object holding two references (p and q) to the matched elements.

Note that some matching links can be derived from the others. For example, the metamodel says that all Person objects must have one 'name' slot. Then as soon as we have objects P and Q matched, their name slots must be automatically matched (the link is thus derived and shown dashed). In contrast, since several phone slots are possible for a person, matching link m_2 between slots $b@A$ and $y@B$ is an independent datum (solid line).

Whatever the way two slots are matched, their matching means that they should hold the same value. If it is not the case, for example, note different numbers in slots $b@A$ and $y@B$, we have a conflict between models. This conflict is represented by setting the value in slot $m_2@M$ to '?' (which is allowed by the metamodel $\boldsymbol{\mathcal{A}}$ in Fig. 9(a)). Note that the metamodel also allows us to skip attribute 'phone', but then we would not have any record of the conflict. By introducing a slot for the conflicting attributes but keeping it empty, we make the conflict explicit and record it in model M. Moreover, two conflicting slots in models A, B can be traced by links $m_2.p^{\mathsf{slt}}$, $m_2.q^{\mathsf{slt}}$. Then we may continue to work with models A, B leaving the conflict resolution for a future processing (as stated by the famous *Living with inconsistencies* principle [12]).

Note that if models were conflicting at their name-attributes, we should resolve this conflict at once because the metamodel in Fig. 9(a) does not allow having null values for names. In this way metamodels can regulate which conflicts can be recorded and kept for future resolution, and which must be resolved right away. Note also that whether two models are in conflict or consistent is determined by the result of their matching, and hence is not a property of the pair itself.

Definition 3. (Osv-model match) Let A, B be two similar osv-models. An *(extensional) model match* is an osv-model M together with two injective model mappings $A \xleftarrow{p} M \xrightarrow{q} B$ (see Fig. 10).

A match is called *complete*, if for any slot $m \in \mathsf{Slot}M$ the following holds:
(*) if $m.p \in \mathsf{Slot}^!A$, $m.q \in \mathsf{Slot}^!B$ and $\mathsf{val}^A(m.p) = \mathsf{val}^B(m.q)$, then $m \in \mathsf{Slot}^!M$.

That is, if a matching slot m links two slots with the same real value, m is not empty (and holds the same value as well by Definition 2).

The term *extensional* refers to the fact that in practice model matches may have some extra (*non-extensional*) information beyond data specified above; we will discuss the issue later in Sect. 4.1. In this section we will say just 'match'.

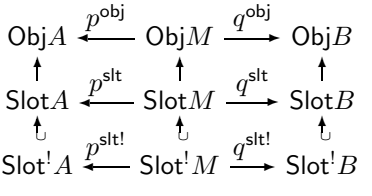


Fig. 10. Matching two osv-models

Completion and consistency of matches. Any incomplete match M can be completed up to a uniquely defined complete match M^* containing M: $\mathsf{Obj}M^* = \mathsf{Obj}M$, $\mathsf{Slot}M^* = \mathsf{Slot}M$, and $\mathsf{Slot}^!M^* \supset \mathsf{Slot}^!M$. We first set $\mathsf{Slot}^!M^* = \mathsf{Slot}^!M$. Then for any slot $m \in \mathsf{Slot}^?M$ we compute two values, $x(m) = \mathsf{val}^A(m.p)$ and $y(m) = \mathsf{val}^B(m.q)$. If x, y are both real values and $x = y$, we move m into set $\mathsf{Slot}^!M^*$ and set $\mathsf{val}_M(m) = x$, otherwise m is kept in $\mathsf{Slot}^?M^*$. Below we will assume that any match is completed.

For a match M and a slot $m \in \mathsf{Slot}^\sharp M$, there are three cases of relationships between values $x(m)$ and $y(m)$ defined above. (Case A): both values are real

but not equal; it means a real conflict between the models. (B): if exactly one of the values is null, say, x, we have an *easy* conflict that can be resolved by propagating real value y from B to A. Let $\mathsf{Slot}^\sharp M \subset \mathsf{Slot}^? M$ denotes the set of slots for which either (A) or (B) holds.

(C) If both values are nulls, the models do not actually conflict although slot m is empty.

Definition 4. (Consistency) Models A and B are called *consistent wrt. their (complete) match* M if set $\mathsf{Slot}^\sharp M$ is empty. (That is, all matched slots either hold a real value or link two empty slots, but the situation of linking two slots with different values is excluded). As a rule, we will say in short that a match M is consistent.

Remark 3. Links in a match can be labeled according to some four-valued logic: no conflict between two real values, no conflict because two nulls, a real conflict (between two real values), and an easy conflict between a value and a null. We leave investigation of this connection for future work.

3.4 Symmetric Deltas and Their Composition

What was described above in terms of matching models understood as replicas, may be also understood in terms of model updates. The following terminology borrowed from category theory will be convenient.

A configuration like $A \xleftarrow{p} M \xrightarrow{q} B$ is called a *model span*: model M is the *head*, models A, B are *feet* and mappings p, q are the *legs* or *projections*. A model span consists of three *set spans*, i.e., spans whose nodes are sets and legs are functions, see Fig. 10. Thus, a (complete) model match is just a (complete) model span whose legs are injections.

Let $A \xleftarrow{p} M \xrightarrow{q} B$ be a complete model span. We may interpret it as an update specification with A and B being the *states* of some fixed model before and after the update. Then elements in sets $\mathsf{Obj}M$ and $\mathsf{Slot}M$ link elements that were kept, A's elements beyond the range of p are elements that were deleted, B's elements beyond the range of q were inserted, and elements from set $\mathsf{Slot}^\sharp M$ (of "conflicting" links) show the attributes that were changed. Now we will call a complete span with injective legs a *(symmetric) delta*, and interpret it as either an (extensional) match or an update.

A delta as specified by Fig. 10 is a symmetric construct, but to distinguish the two models embedded into it, we need to name them differently. Say, we may call model A the *left* or better the *source*) model, and model B the *right* or better the *target* model. It is suggestive to denote a delta by an arrow $\mathbf{\Delta} : A \Rightarrow B$, whose double-body is meant to remind us that a whole triple-span diagram (Fig. 10) is encoded. The same diagram can be read in the opposite direction from the right to the left, which means that delta $\mathbf{\Delta}$ can be inverted into delta $\mathbf{\Delta}^{-1} : B \Rightarrow A$ (see Appendix B, p. 151 for a precise definition).

Suppose we have two consecutive deltas

$$A \overset{\mathbf{\Delta}_1}{\Longrightarrow} B \overset{\mathbf{\Delta}_2}{\Longrightarrow} C \text{ with } \mathbf{\Delta}_1 = (A \xleftarrow{p_1} M_1 \xrightarrow{q_1} B) \text{ and } \mathbf{\Delta}_2 = (B \xleftarrow{p_2} M_2 \xrightarrow{q_2} C)$$

between Ann's, Bob's and, say, Carol's models. To compose them, we need to derive a new delta $A \overset{\boldsymbol{\Delta}}{\Longrightarrow} C$ from deltas $\boldsymbol{\Delta}_1$ and $\boldsymbol{\Delta}_2$.

Since deltas are complete spans, each of them is determined by two set spans, Δ_i^{obj} and Δ_i^{slt}, $i = 1, 2$, which can be sequentially composed. The reader may think of deltas as representations of binary relations, and their composition as the ordinary relational composition $\bowtie$; a precise formal definition of delta composition via the so called *pullback* operation is in Appendix B, p. 152.

In this way we derive a new osv-model N determined by sets $\mathsf{Obj}N \overset{\text{def}}{=} \mathsf{Obj}M_1 \bowtie \mathsf{Obj}M_2$ and $\mathsf{Slot}N \overset{\text{def}}{=} \mathsf{Slot}M_1 \bowtie \mathsf{Slot}M_2$, and by function $\mathsf{obj}_N : \mathsf{Slot}N \to \mathsf{Obj}N$ defined in the natural way (via the universal property of pullbacks; this is where the categorical formulation instantly provides the required result). Projections are evident and thus we have two set spans $\boldsymbol{\Delta} = (A \overset{p^{\mathsf{x}}}{\leftarrow} \mathsf{x}N \overset{q^{\mathsf{x}}}{\to} C)$ with $\mathsf{x} = \mathsf{obj}, \mathsf{slt}$. These data give us a span N with empty set $\mathsf{Slot}^!N$. However, we can complete N as described above (we let N denote the completion too), and so obtain a new delta $\boldsymbol{\Delta} = (A \overset{p}{\leftarrow} N \overset{q}{\to} C)$ between models. Associativity of so defined composition follows from associativity of span composition (Appendix B). In addition, a complete span $A \leftarrow A \to A$ whose legs consist of identity functions between sets is a unit of composition. We have thus proved

Theorem 1. *The universe of osv-models and symmetric deltas between them is a category.*

Exercise 1. Explain why $\mathsf{Slot}^!N \supseteq \mathsf{Slot}^!M_1 \bowtie \mathsf{Slot}^!M_2$ but equality does not necessarily hold.

4 Simple Update Propagation, I: Synchronizing Replicas

By a replica we understand a maintained copy of a model, and assume that replication is *optimistic*: replicas are processed independently and may conflict with each other, which is optimistically assumed to appear infrequently [19]. Then it makes sense to record conflicts to resolve them later, and continue to work with only partially synchronized replicas. The examples considered in Section 2 (Fig. 3) are simple instances of replica synchronization. We have considered them in a *concrete* way by looking inside models and their mappings. The present section aims to build an abstract algebraic framework in which models and mappings are treated as indivisible points and arrows.

Subsection 4.1 introduces the terminology and basic notions of replica synchronization; there is an overlap with the previous section that renders the present section independent. Subsection 4.2 develops a basic intuition for the algebraic approach to modeling synchronization. Subsections 4.3 and 4.4 proceed with algebraic modeling as such: constructing algebraic theories and algebras (instances of theories).

4.1 Setting the Stage: Delta × Delta = Tile

Terminology. There are two main types of representations for model differences: operational and structural, which are usually called *directed* and *symmetric* deltas respectively [20]. The former is basically a sequence (log) of edit operations: add, change, delete (see, e.g., [21]). The latter is a specification of similarities and differences of the two models ([22]).

A symmetric delta can be seen *extensionally* as a family of matching links, in fact, as a binary relation; in the previous section we formalized symmetric deltas as *(complete) spans* (reified binary relations). Besides extension, a symmetric delta may contain *non-extensional* information: matching links can be annotated with authorship, time stamps, update propagation constraints and the like. We also call deltas *mappings* and denote them by arrows (even symmetric deltas, see Sect. 3.4).

Now suppose we have two replicas of the same model maintained by our old friends Ann and Bob, Fig. 11. Nodes A, B are snapshots of Ann's and Bob's replica at some time moment, when we want to compare them. The horizontal arrow m denotes a relationship — match— between the replicas. We interpret matches as symmetric deltas (spans) with, perhaps, some additional (non-extensional) data. Ann and Bob work independently and later we have two updated versions A' and B' with arrows a and b denoting the corresponding updates. We may interpret updates structurally as symmetric deltas. Or we may interpret them operationally as directed deltas (edit logs).

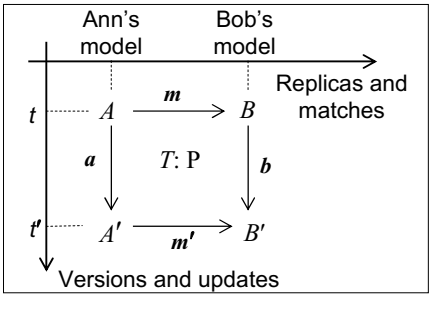


Fig. 11. The space of model versioning

The four deltas m, a, m', b are mutually related by *incidence relationships*: $\partial_s m = \partial_s a, \partial_t m = \partial_s b$, etc. (where $\partial_s x, \partial_t x$ denote the source and the target of arrow x), and together form a structure that we call a *tile*. The term is borrowed from a series of work on behavior modeling [23], and continues the terminological tradition set up by the Harmony group's *lenses* — naming synchronization constructs by geometric images.

Visually, a tile is just a square formed by arrows with correspondingly sorted arrows. To avoid explicit sorting of arrows in our diagrams, we will always draw them with updates going vertically and matches horizontally. A tile can be optionally labeled by the name of some tile's property (predicate) P. Expression $T{:}P$ means that T has property P, i.e., $T \models P$ or $T \in [\![P]\!]$ with $[\![P]\!]$ denoting the extension of P. The name of the tile may be omitted but the predicate label should be there if $T \models P$ holds.

The tile language: matches vs. updates. To keep the framework sufficiently general, we do not impose any specific restrictions on what matches and updates really are, nor do we assume that they are similar specifications. For example,

matches may be annotated with some non-extensional information that does not make sense for updates, e.g., priorities of update propagation (say, 'name' modifications are propagated from Ann's model to Bob's, while 'phones' are not) or "matching ranks" (how much we are sure that elements $e@A$ and $e'@B$ are the same, see [24] for a discussion). Furthermore, we may have matches defined structurally (with annotations or not) whereas updates operationally.

Therefore, we do not suppose that matches can be sequentially composed with updates (and vice versa). But of course updates can be composed with updates, and matches with matches, although match composition can be non-trivial, if at all well-defined, because of non-extensional information. For example, let $m^+: X \to Y$ denotes a match consisting of symmetric delta (relation, span) m augmented with some non-extensional information. For two consecutive matches $m_1^+: A \to B$, $m_2^+: B \to C$, their extensional parts can be composed as relations producing delta $m = m_1; m_2: A \to C$, but to make m into a match m^+ we need to compose somehow non-extensional parts of the matches. We leave the issue for the future work and in this paper will not compose matches.

The situation with updates is simpler. Either they are interpreted structurally as symmetric deltas (spans), or operationally as edit logs, they are sequentially composable in the associative way. For symmetric deltas it is shown in Sect. 3.4; and it is evident for edit logs (whose composition is concatenation).

In addition, we assume that for every model A there are an *idle update* $\mathbb{1}^{\mathsf{b}}_A: A \to A$ that does nothing, and an *identity match* $\mathbb{1}^{\mathsf{h}}_A: A \to A$ that identically matches model A to itself. For the structural interpretation of arrows, both idle updates and identity matches are nothing but spans whose legs are identity mappings (and no extra non-extensional information is assumed for matches). For the operational interpretation, idle/identity arrows are empty edit logs.

Thus, in the abstract setting we have a structure consisting of two reflexive graphs, ***Modmch*** of models and *matches*, and ***Modupd*** of *models* and *updates*, which share the same class of objects ***Mod*** but have different arrows. Moreover, arrows in graph ***Modupd*** are composable (associatively) and ***Modupd*** is a category. We will call such a structure a *1.5-sorted category* and denote it by $\mathbb{M}$**od** (if ***Modmch*** also were a category, $\mathbb{M}$**od** would be a *two-sorted category*)(see Sect. B).

Simple synchronization stories via tiles. Despite extreme simplicity of the language introduced above, it allows us to describe some typical replication situations as shown in Fig. 12. The diagrams in the figure can be seen as specifications of use cases ("stories") that have happened, or may happen, in some predefined context. The meaning of these stories is easily readable and explained in the captions of the diagrams (a-d). In diagram (b), symbol $\cong$ denotes the predicate of being an *isomorphic* match (i.e., we assume that a subclass $[\![\cong]\!]$ of arrows in graph ***Modmch*** is defined).

The stories could be made more interesting if we enrich our language with diagram predicates, say, P_{h} and P_{v}, allowing us to compare matches and updates. Then, for example, by declaring that tile T belongs to the class $[\![P_{\mathsf{h}}]\!]$ (as shown by diagram (c)*), we say that match m' is "better" than m. Such predicates can be

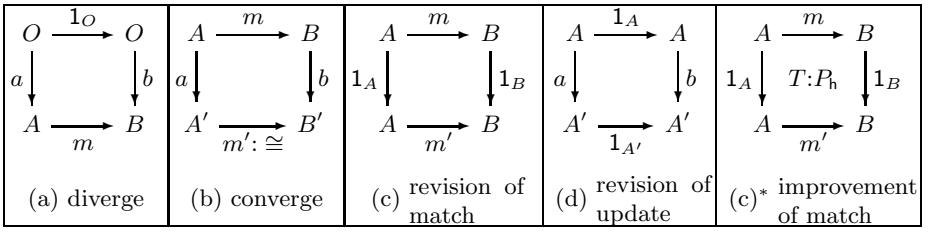


Fig. 12. Several replication stories via tiles

seen as arrows between arrows, or *2-arrows*, and give rise to a rich framework of *2-categories* and *bicategories* (see, e.g., [25]). We leave this direction of modeling replication for future work.

The "historian's" view on synchronization scenarios, even with comparison predicates, is not too interesting. The practice of model synchronization is full of automatic and semi-automatic operations triggered automatically or by the user's initiation. Thus, we need to enrich our language with synchronization operations.

4.2 Update Propagation via Algebra: Getting started

As discussed in Sect. 2.2, algebraic operations modeling synchronization procedures should be *diagrammatic*: they take a configuration (diagram) of matches and updates that conform to a predefined *input* pattern, and add to it new matches and updates conforming to a predefined *output* pattern. These new elements are to be thought of as computed or derived by the operation. In this section we consider how diagram operations work with a typical example, and develop a basic intuition about the algebraic approach to modeling synchronization.

Update propagation: A sample diagram operation. Propagating updates from one replica to another is an important synchronization scenario. We model it by diagram operation fPpg shown in Fig. 13(a). The operation takes a match m between replicas and an update a of the source replica, and produces an update b of the target replica and a new match m'. The input/output arrows are shown by solid/dashed lines resp.; the direction of the operation is shown by the doubled arrow in the middle. (To be consistent, we should also somehow decorate node B' but we will not so so.)

We write $(b, m') = \mathsf{fPpg}(a, m)$ and call the quadruple of arrows (tile) $T = (a, m, b, m')$ an *application instance* of the operation. Other pairs of input arrows will give other application instances of the same operation; hence, notation T:fPpg. (The name T is omitted in the diagram). This notation conforms to labeling tiles by predicates introduced earlier. Operation fPpg defines a predicate fPpg^* of square shape: for a quadruple of arrows (a, m, b, m') forming a square, we set $\mathsf{fPpg}^*(a, m, b, m')$ is true iff $(b, m') = \mathsf{fPpg}(a, m)$; in this case we say that the quadruple (a, m, b, m') is a fPpg-tile. Later we will omit the star superindex.

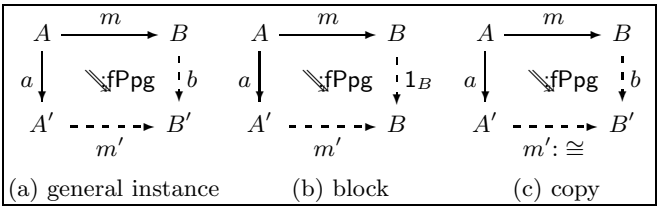


Fig. 13. Forward update propagation (a) and its two special cases (b,c)

Below we will also use the dot-notation for function applications, $(b, m') = (a, m).\mathsf{fPpg}$ to ease reading complex formulas. Since the operation produces two elements, we need special *projection* operations, upd and mch, that select the respective components of the entire output tuple: $b = (a, m).\mathsf{fPpg.upd}$ and $m' = (a, m).\mathsf{fPpg.mch}$.

Update policies and algebra. There are two extreme cases of update propagation with fPpg.

One is when nothing is propagated and hence the output update is idle as shown in diagram Fig. 13(b). Then propagation amounts to *rematching*: updating the match from m to m'. If this special situation, i.e., equality $(a, m).\mathsf{fPpg.upd} = 1_B$, holds for any update a originating at m's source, we have a very strong propagation policy that actually *blocks* replica B wrt. updates from A.

The opposite extremal case is when the entire updated model is propagated and overwrites the other replica as shown in diagram (c). A milder variant would be to propagate the entire A' but not delete the unmatched part of B, then match m' would be an embedding rather than isomorphism.

In-between the two extremes there are different propagation policies as discussed in Sect. 2.2.1. The possibility of choice is in the nature of synchronization problems: as a rule, some fragments of information are missing and there are several possible choices for model B'. To make computation of model B' deterministic, we need to set one or another propagation policy. Yet as soon as a policy is fixed, we have an algebraic operation of arity shown in Fig. 13(a). Thinking algebraically, a policy *is* an operation (cf. Discussion in Sect. 2.2.1).

Remark 4. So-called *universal* properties and the corresponding operations (see Appendix A.1) are at the heart of category theory. It explains attempts to model update policies as universally defined operations [26]. However, our examples show that, in general, a propagation policy could not be universally defined simply because many policies are possible (while universally defined operations are unique up to isomorphism).

Algebra: action vs. "history". The mere assertion that some components of a story specified by a tile are derived from the other components may be a strong statement. Let us try to retell our simple synchronization stories in Fig. 12 in an algebraic way.

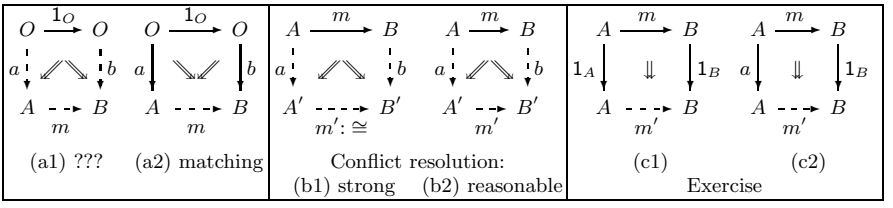


Fig. 14. Replication stories and algebra: Fig. 12 processed algebraically

Diagram Fig. 14(a1) says that three arrows (a, m, b) are produced by applying some operation to the identity match, that is, in fact, to model O. This is evidently meaningless because triple (a, m, b) cannot be derived from O alone. In contrast, diagram (a2) is a reasonable operation: given two updates of the same source, a match between them can be computed based on the information provided by the input data.

Diagram (b1) says that any two matched replicas can be made isomorphic. It is a very strong statement: we assume that all conflicts can be resolved, and differences between replicas can be mutually propagated in a coherent way. A more reasonable algebraic model of conflict resolution is specified by diagram (b2): the result of the operation is just another match m' presumably better (with less conflicts) than m. Augmenting the language with constructs formally capturing the meaning of "better" (e.g., 2-arrows) would definitely be useful, and we leave it for future work.

Exercise 2 (*). Diagrams (c1,c2) present two algebraic refinements of the synchronization story specified in Fig. 12(c). Explain why diagram (c1) does not make much sense whereas (c2) specifies a reasonable operation. *Hint*: Note an important distinction of diagram (c1) from diagram (b2).

4.3 An Algebraic Toolbox for a Replica Synchronization Tool Designer

Suppose we are going to build a replica synchronization tool. Before approaching implementation, we would like to specify what synchronizing operations the tool should perform, and what behavior of these operations the tool should guarantee; indeed, predictability of synchronization results is important for the user of replication/versioning tools (cf. [3]). Hence, we need to fix a signature of operations and state the laws they must obey; in other words, we need to fix a suitable *algebraic theory*. The tool itself will be an *instance* of the theory, that is, an *algebra*: sorts of the theory will be interpreted by classes of replicas the tool operates on, and operations will be interpreted by actual synchronization procedures provided by the tool.

Two main ingredients constituting an algebraic theory are a *signature* of operations with assigned arity shapes, and a set of *equational laws* prescribing the intended behavior of the operations. In ordinary algebra, operation arities are

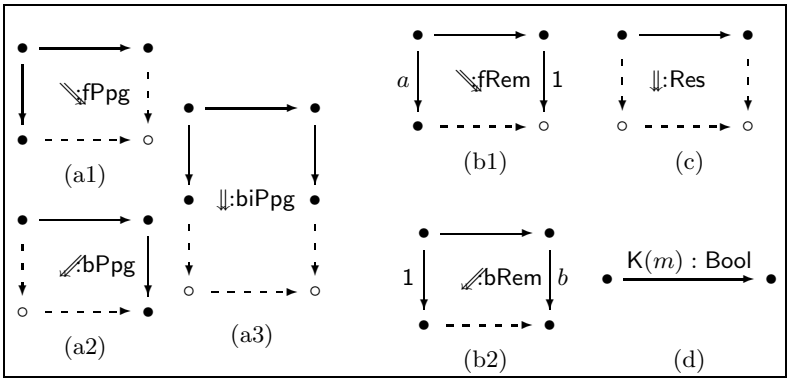


Fig. 15. Replica synchronization operations: update propagation (a), rematching (b), conflict resolution (c), and Boolean test for consistency of matches (d)

sorted sets; in diagram algebra, arities are sorted graphs but the principal ideas and building blocks remain the same. In this section we specify a pool of diagram operations for modeling synchronization procedures, and a pool of laws that they should, or may want, to satisfy. Together they are meant as an algebraic toolbox with which a tool designer can work.

The carrier structure. All our operations will be defined over 1.5-sorted categories, i.e., two-sorted reflexive graphs with arrows classified into horizontal (matches) and vertical (updates); the latter are composable and form a category.

Operations. A precise definition of a diagram operation over a two-sorted graph is given in Sect. B. For the present section it is sufficient to have a semi-formal notion described above.Recall that in order to avoid explicit sorting of arrows in our diagrams, we draw them with geometrically vertical/horizontal arrows being formally vertical/horizontal.

Figure 15 presents a signature of operations intended to model synchronization procedures. The input/output arrows are distinguished with solid/dashed lines, and input/output nodes are black/white.

Diagrams Fig. 15(a1,a2) show operations of *forward* and *backward* update propagation. The former was just considered; the latter propagates updates against the direction of match and is a different operation. For example, if the replica at the source is in some sense superior to the replica at the target, forward propagation may be allowed to propagate deletions whereas the backward one is not. Diagram (a3) specifies bi-directional update propagation. It takes a match and two parallel updates and mutually propagates them over the match; the latter is then updated accordingly.

Diagrams Fig. 15(b1,b2) show operations of forward and backward *rematching*. If for a given match $m\colon A \to B$, one of the replicas, say, A, is updated, we may want to recompute the match but do not change the other replica B. This scenario is modeled by operation fRem in Fig. 15(b1), where the update of the other replica is set to be idle. Thus, operation fRem actually has two arguments (the left update

and the upper match) and produces the only arrow — an updated match (at the bottom). The backward rematch works similarly in the opposite direction. The operation of bidirectional rematching does not make sense (Exercise 1 above). If we were modeling both matches and updates by relations (spans), then rematch would nothing but sequential span composition Sect. 3.3. However, as we do not compose updates and matches, we model their composition by a special tile operations.

Finally, Fig. 15(c) specifies operation Res of *conflict resolution*. It takes a match between two replicas that, intuitively, may be inconsistent, and computes updates a, b necessary to eliminate those conflicts that can be resolved automatically without user's input.

Other synchronization operations are possible, and the signature described above is not intended to be complete. Neither is it meant to be fully used in all situations. Rather, it is a pool of operations from which a tool designer may select what is needed.

Predicates. To talk about consistency of matches, we need to enrich our language with a *consistency* predicate (think of strongly consistent matches from Sect. 3.3).

Diagram (d) presents it as a Boolean-valued operation: for any match m a Boolean value is assigned, and we call m *consistent* if $\mathsf{K}(m) = 1$. (The letter K is taken from "*K*onsistency": denoting the predicate by C would better fit the grammar but be confusing wrt. terms Classes and Constraints.) In our diagrams we will write $m{:}\mathsf{K}$ for $\mathsf{K}(m) = 1$. Semantically, we have a class of consistent matches $\mathbf{K} = \{m\colon\ \mathsf{K}(m) = 1\}$.

Remark 5. Consistency is often considered as a binary predicate K' on models: replicas (A, B) are *consistent* if $\mathsf{K}'(A, B)$ holds [6]. Our definition is essentially different and moves the notion of consistency from pairs of replicas to matches. Indeed, as discussed in sections 2.2, 3.3, multiple matches between replicas are possible, and it is a match $m\colon A \to B$ that makes the pair (A, B) consistent or inconsistent.

Remark 6. The presence of predicates makes our theory non-algebraic. A standard way to bring it back to algebra is to define predicates via equations between operations, if it is possible. Another approach is to work in the framework of order-sorted algebra [27].

Equational laws. Equations the operations must satisfy are crucial for algebraic modeling. Without them, algebraic theories would define too broad classes of algebras encompassing both adequate and entirely inadequate algebraic models.

Equational laws for diagram operations can be concisely presented by diagrams as well. Consider, for example, diagram Fig. 16(a1), whose arrows are labeled by names (identifiers) of matches and updates. The names express the following equation: for any match m, $\mathsf{fPpg}(1_{\partial_s m}, m) = (1_{\partial_t m}, m)$. This is a general mechanism: if all arrows in the tile have different names, the tile specifies a generic instance of the operation without any restrictions, but the presence of common names amounts to equational constraints like above.

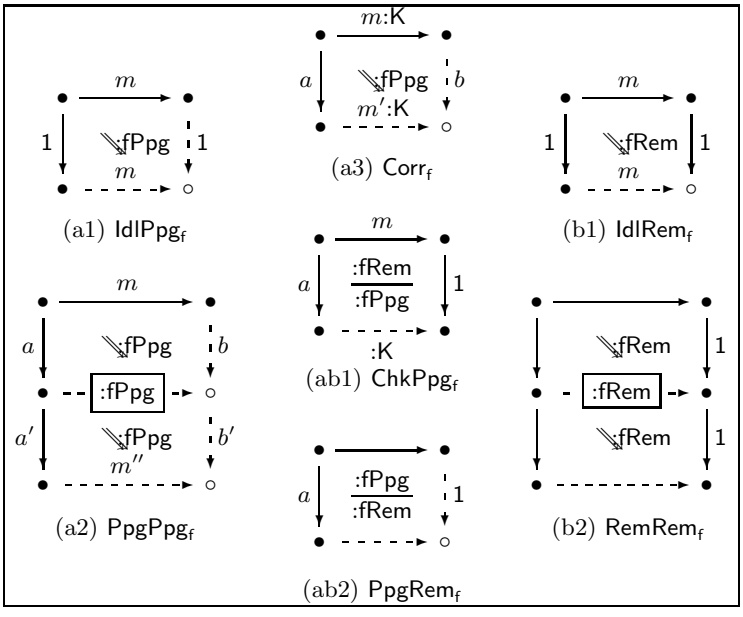


Fig. 16. Replica synchronization: the laws

The equation expressed by Fig. 16(a1) has a clear interpretation: given a match m, the idle update on the source is propagated into the idle update on the target while the match itself is not changed. We call the law $\mathsf{IdlPpg_f}$ following a general pattern of naming such laws by concatenating the operation names (take the idle update and propagate it; index f refers to forward propagation). The pattern was invented by the Harmony group for lenses and turned out very convenient.

Diagram Fig. 16(a2) displays two fPpg-tiles vertically stacked (ignore the boxed label for a while). It means that the output match of the upper application of fPpg is the input match for the lower application. Since updates are composable, the outer rectangle in the diagram is also a tile whose updates are $a; a'$ and $b; b'$. Now the boxed label says that the outer tile is also an application instance of fPpg. (In more detail, given a match m and two consecutive updates a, a' on its source, we have $\mathsf{fPpg}(a; a', m) = (b; b', m'')$ where $(b, m') = \mathsf{fPpg}(a, m)$ (name m' is hidden in the diagram) and $(b', m'') = \mathsf{fPpg}(a', m')$.) We will phrase this as follows: if the two inner tiles are fPpg, then the outer tile is also fPpg (note also the name of the law). Thus, composed updates are propagated componentwise.

Diagram Fig. 16(a3) says that if $(b, m') = \mathsf{fPpg}(a, m)$ and $m \in \mathsf{K}$, then $m' \in \mathsf{K}$ as well: consistency of matches is not destroyed by update propagation. We call an update propagation *correct* if it satisfies this requirement, hence the name of the law. Note the conditional nature of the law: it says that the resulting match is consistent if the original match is consistent but does not impose any obligations if the original match is inconsistent. This formulation fixes the problem of the unconditional correctness law stated in [6].

Exercise 3. Explain the meaning of diagrams (b1) and (b2) in Fig. 16

Now we consider laws regulating interaction between the two operations. The law specified by diagram (ab1) is conditional. The argument of the premise and the conclusion is the entire tile, and the diagram says: if a tile is an instance of fRem with output match satisfying K, then the tile is also an instance of fPpg. Formally, $\mathsf{fRem}^*(T)$ implies $\mathsf{fPpg}^*(T)$ for a tile of the shape shown in the diagram (recall that starred names denote predicates defined by operations). That is, if $m' = \mathsf{fRem}(a, m) \in \mathsf{K}$ then $\mathsf{fPpg}(a, m) = (m', 1_{\partial_t m})$. The meaning of the law is that if we update the source, and the updated match m' is consistent, then nothing should be propagated to the target. This is a formal explication of the familiar requirement on update propagation: "first check, then enforce" (cf. Hippocraticness in [6]). Hence the name of the law, ChkPpg.

Exercise 4. Explain the meaning of diagram (ab2) Fig. 16

Exercise 5 (*). Formulate some laws for the operation of conflict resolution, and specify them diagrammatically.

There is no claim that the set of laws we have considered is complete: other reasonable laws can be formulated. The goal was to show how to specify equational laws, and how to interpret them, rather then list them "all".

4.4 Replica Synchronization Tools as Algebras

In this section, we build a simple algebra intended to model a replica synchronization tool as it was explained at the beginning of Sect. 4.3.

We first fix a theory (= signature + laws). For the signature, we take four operations (to be precise, operation symbols) $(\mathsf{fPpg}, \mathsf{bPpg}, \mathsf{fRem}, \mathsf{bRem})$ with arities specified in Fig. 15. These operations can be interpreted over any 1.5-sorted category encompassing any number of replicas. However, we assume that our synchronization tool will only work with two replicas propagating updates from one to the other and back. Hence, we need to specify a specific 1.5-sorted category adequate to our modest needs.

Definition 5. A *(binary) replication lane* $\mathbf{r}$ is given by the following data.

(a) Two categories, $\mathbf{A}$ and $\mathbf{B}$, whose objects are called *replicas* (or *models*), and arrows are *updates*. (For a category $\mathbf{X}$, its classes of objects and arrows are denoted by, resp., $\mathbf{X}_0$ and $\mathbf{X}_1$.) Specifically, objects of $\mathbf{A}$ are called *source* replicas and those of $\mathbf{B}$ the *target* ones.

(b) A set $\mathbf{M}$ whose elements are called *matches* from $\mathbf{A}$- to $\mathbf{B}$-replicas, and two functions (*legs*), $\mathbf{A}_0 \overset{\partial_s}{\leftarrow} \mathbf{M} \overset{\partial_t}{\rightarrow} \mathbf{B}_0$, from matches to replicas. If for a match $m \in \mathbf{M}$, $\partial_s(m) = A$, $\partial_t(m) = B$, we write $m\colon A \to B$.

(c) A set $\mathbf{K} \subset \mathbf{M}$ of *consistent* matches.

Figure 17 visualizes the definition: updates are vertical, and matches are horizontal or slanted (solid or dotted-dashed for being consistent or inconsistent).

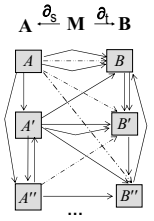


Fig. 17. Replica lane

We denote a replication lane by a bulleted arrow $\mathbf{r} : \mathbf{A} \bullet\!\!\rightarrow \mathbf{B}$. If replicas are considered within the same versioning space, categories $\mathbf{A}$ and $\mathbf{B}$ coincide, and we call the lane *unary*, $\mathbf{r}$: $\mathbf{A} \bullet\!\!\rightarrow \mathbf{A}$.

Now we define an algebra over a replica lane.

Definition 6. A *diagonal replica synchronizer* is a pair $\delta = (\mathbf{r}^\delta, \Sigma^\delta_{\mathsf{brSync}})$ with $\mathbf{r}^\delta$ a replica lane and $\Sigma^\delta_{\mathsf{brSync}} = (\mathsf{fPpg}^\delta, \mathsf{bPpg}^\delta, \mathsf{fRem}^\delta, \mathsf{bRem}^\delta)$ a quadruple of diagram operations over $\mathbf{r}^\delta$ of the arities specified in Fig. 15. The name *diagonal* refers to the fact that propagation operations act along diagonals of operation tiles, and bidirectional propagation (for parallel updates) is not considered.

It is convenient to denote a replica synchronizer by an arrow $\delta : \mathbf{A} \bullet\!\!\rightarrow \mathbf{B}$ whose source and target refer to the source and target of the replica lane $\mathbf{r}^\delta$.

A diagonal synchronizer is called *well-behaved (wb)* if the pair $(\mathsf{fPpg}, \mathsf{fRem})$ satisfies the laws $\mathsf{IdlPpg_f}$,$\mathsf{Corr_f}$,$\mathsf{IdlRem_f}$,$\mathsf{ChkPpg_f}$ specified in Fig. 16, and the pair $(\mathsf{bPpg}, \mathsf{bRem})$ satisfies the backward counterparts of those laws. A wb diagonal synchronizer is called *very well-behaved (vwb)* if the laws $\mathsf{PpgPpg_f}$,$\mathsf{RemRem_f}$ and their backward counterparts hold too.

Modularization of the set of laws provided by the notions of wb and very wb synchronizer is somewhat peculiar from the categorical standpoint because it joins unitality (preservation of units of composition, ie, idle updates) with other laws but separates it from compositionality, and the very terms are not very convenient. However, this modularization and terminology follow the terminology for lenses [2] and make comparison of our framework with lenses easier (see [28] for an analysis of these laws in the discrete setting).

The definition above is intuitively clear but its precise formalization needs a careful distinction between syntax and semantics of a diagram operation, see Sect. B.

5 Simple Update Propagation II: Forward and Backward View Maintenance

In this section we consider synchronization of a source model and its view. The content is parallel to replica synchronization and the algebraic model is developed along the same lines. Yet view synchronization is essentially different from replica synchronization.

5.1 View vs. Replica Synchronization

Examples in Sect. 2.2 and Appendix C show that a view definition can be modeled by a metamodel mapping $\boldsymbol{\mathcal{S}} \xleftarrow{\boldsymbol{v}} \boldsymbol{\mathcal{T}}$ that sends elements of the view (target) metamodel $\boldsymbol{\mathcal{T}}$ to basic or derived elements of the source metamodel $\boldsymbol{\mathcal{S}}$.[4] In addition, the mapping must be compatible with the structure of the metamodels

[4] Derived elements of $\boldsymbol{\mathcal{S}}$ are, in fact, queries against $\boldsymbol{\mathcal{S}}$ seen as a data schema.

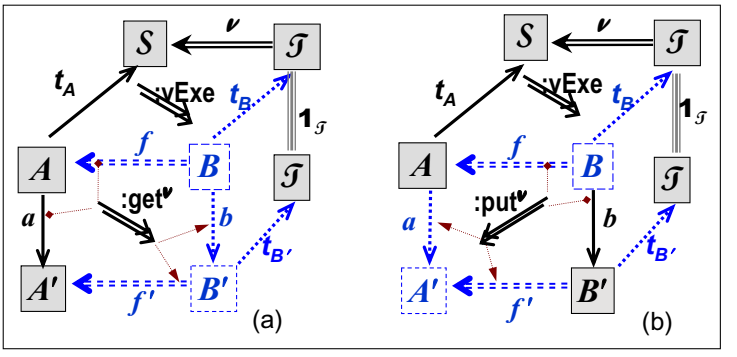


Fig. 18. View maintenance: Forward (a) and backward (b) update propagation

(and send a class to a class, an attribute to an attribute *etc.*) Such a view definition can be executed for any instance A of $\boldsymbol{\mathcal{S}}$, and produce a $\boldsymbol{v}$-view of A, i.e., a $\boldsymbol{\mathcal{T}}$-instance denoted by $A{\restriction_{\boldsymbol{v}}}$, along with a traceability mapping $A \xleftarrow{\overline{\boldsymbol{v}_A}} A{\restriction_{\boldsymbol{v}}}$ (see Sect. C for details).[5] In fact, we have a diagram operation specified by the top face of cube (a) in Fig. 18, where $B = A{\restriction_{\boldsymbol{v}}}$ and $f = \overline{\boldsymbol{v}_A}$.

If the source A is updated, the update is propagated to the view by operation $\mathsf{get}^{\boldsymbol{v}}$ ("getView") shown in the front face of the cube Fig. 18(a). The operation takes a source update a and view mapping f, and produces a view update b together with a new view traceability mapping f'. A reasonable requirement is to have $f' = \overline{\boldsymbol{v}_{A'}}$ and $B' = A'{\restriction_{\boldsymbol{v}}}$. In the database literature, such operations have been considered as *view maintenance* [29].

If the view is updated via $b\colon B \to B'$ (the front face of cube Fig. 18(b)), we need to update the source correspondingly and find an update $a\colon A \to A'$ such that $B' = A'{\restriction_{\boldsymbol{v}}}$; simultaneously, a new traceability mapping $f' = \overline{\boldsymbol{v}_{A'}}$ is computed. Since normally a view abstracts away some information, many updates a may satisfy the condition. To achieve uniqueness, we need to consider additional aspects of the situation (metamodels, view definition, the context) — this is the infamous view update problem that has been studied in the database literature for decades [30]. Yet we assume that somehow an update propagation policy ensuring uniqueness is established, and hence we have an operation $\mathsf{put}^{\boldsymbol{v}}$ ("put update back") specified by the front face of the cube. Names 'get' and 'put' are borrowed from the lens framework [2], but in the latter neither update nor view mappings are considered. Also, lenses' operation get corresponds to our vExe_0.

Despite similar arity shapes of bidirectional pairs (get,put) in view synchronization and (fPpg,bPpg) in replica synchronization, the two tasks are different.

First we note that in the view update situation, consistency relation $\mathbf{K}$ can be derived rather than independently postulated: we set

$$\mathbf{K} \stackrel{\text{def}}{=} \left\{ A \xleftarrow{f} B\colon\ f = \overline{\boldsymbol{v}_A} \right\}. \tag{Cons}$$

[5] View $A{\restriction_{\boldsymbol{v}}}$ can be seen as $\boldsymbol{v}$-*projection* of model A to space of $\boldsymbol{\mathcal{T}}$-models, hence symbol $\restriction$ denoting restriction.

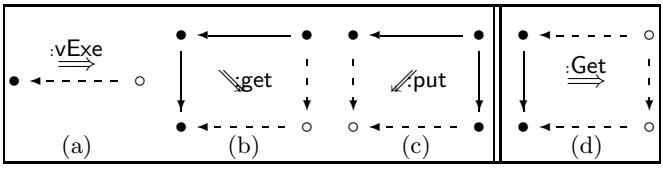


Fig. 19. View synchronization: the signature

Next we assume that the view is entirely dependent on the source: once the source is updated, the view is automatically recomputed so that the source update does not create inconsistency. On the other hand, if the view is updated, it at once becomes inconsistent with the source since only one view corresponds to the source. Hence, there is no need for the "first-check-then-enforce" principle, and any view update must be propagated back to the source to restore consistency.

The result is that in contrast to replica synchronization, it is reasonable to assume that view update propagation always acts on consistent matches as shown by the front faces of cubes in Fig. 18(a,b), and produces consistent matches. We may thus ignore inconsistent matches completely. It implies that the correctness and "first-check-then-enforce" laws of replica synchronization become redundant, and we do not need rematching operations. This setting greatly simplifies the theory of update propagation over views. The rest of the section described the basics of such a theory.

5.2 The Signature and the Laws

Figure 19 (a,b,c) presents arity shapes of the three operations we will consider. As before, the input nodes and arrows are black and solid, the output ones are white and dashed. The meaning of the operations is clear from the discussion above. Operation (d) will be discussed later.

Figure 20 specifies some laws the three operations must satisfy. The laws IdlGet, IdlPut, GetGet, and PutPut in cells (b1,b2,c1,c2) are quite similar to the respective laws for forward and backward propagation discussed in Sect. 4. They say that idle updates on one side result in idle updates on the other side, and composition of updates is propagated componentwise.

The PutGet law in cell (bc) states that any put-tile is automatically a get-tile. In the string-based notation, if $(a, f') = \mathsf{put}(b, f)$ then $(b, f') = \mathsf{get}(a, f)$.

The Exe! law in cell (a!) states that any match (the empty premise) is a correct view traceability mapping produced by vExe applied to the target of the match. This implies that put and get only apply to correct matches as discussed above. We could a priori postulate this, and rearrange operation get into operation Get specified in Fig. 19(d), which both computes the views and propagates updates. It is a possible way to go (cf. the functorial approach to the view update problem [26]), but this paper explores a different setting, in which vExe computes the view model only and get propagate updates using view traceability mappings.

Exercise 6. Formulate the horizontal counterparts of GetGet and PutPut, and explain their meaning. *Hint*: consider a composed view definition in Fig. 5.

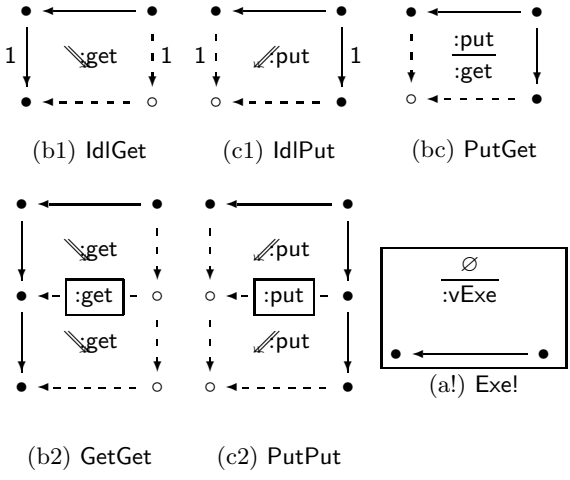


Fig. 20. View synchronization: the laws

5.3 View Synchronization

Definition 6. A *view lane* **v** is given by the following data.

(a) Two categories $\mathbf{A}$ and $\mathbf{B}$, whose objects are called *models* and arrows are *updates*. Objects of $\mathbf{A}$ are called *source* models and those of $\mathbf{B}$ are *views*.

(b1) A span of sets, $\mathbf{A}_0 \xleftarrow{\partial_\mathsf{t}} \mathbf{V} \xrightarrow{\partial_\mathsf{s}} \mathbf{B}_0$ with ∂_t and ∂_s being total functions giving the *target* and the *source* for each *view traceability* mapping $f \in \mathbf{V}$. We write $A \xleftarrow{f} B$ if $\partial_\mathsf{t}(f) = A$ and $\partial_\mathsf{s}(f) = B$.

(b2) An operation $\mathsf{vExe} : \mathbf{A}_0 \to \mathbf{V}$ of *view execution* such that for any model $A \in \mathbf{A}_0$ and any mapping $v \in \mathbf{V}$, the following two laws hold:

(ExeDir) $\quad \partial_\mathsf{t}\mathsf{vExe}(A) = A$

(Exe!) $\quad$ if $\partial_\mathsf{t} v = A$, i.e., $A \xleftarrow{v} B$, then $v = \mathsf{vExe}(A)$

Thus, for any $A \in \mathbf{A}_0$ we have a unique traceability mapping $A \xleftarrow{\mathsf{vExe}(A)} B$ targeting A, and any traceability mapping is of this form.

We denote the composition $\mathsf{vExe}; \partial_\mathsf{s}$, which gives the source of the arrow $\mathsf{vExe}(A)$, by vExe_0. Then, given a source A, its view $B = \mathsf{vExe}_0(A)$.

Evidently, $A \neq A'$ implies $\mathsf{vExe}(A) \neq \mathsf{vExe}(A')$, but it may happen that $B = \mathsf{vExe}_0(A) = \mathsf{vExe}_0(A')$ for different A, A' because view abstracts away some information.

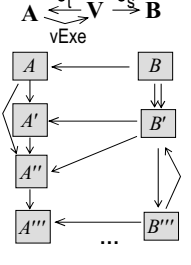


Fig. 21. View lane

Fig. 21 visualizes the definition: updates are vertical arrows, and view traceability mappings are horizontal. (Compare this figure with Fig. 17 and note the difference between the carrier structures for replication and view updates.)

Definition 8. A *view synchronizer* over a lane $\mathbf{v}$ is a pair of diagram operations $\lambda = (\mathsf{get}, \mathsf{put})$ whose arities are specified in Fig. 19. Notation λ reminds us *l*enses.

We will denote view synchronizers by arrows $\lambda\colon \mathbf{A} \to \mathbf{B}$ and write $\partial_{\mathsf{s}}\lambda$ for $\mathbf{A}$ and $\partial_{\mathsf{t}}\lambda$ for $\mathbf{B}$. With this notation, operations get of forward view maintenance and vExe of view computation go in the direction of arrow λ whereas the backward operation put goes in the opposite direction. Thus, although vExe computes from $\mathbf{A}$ to $\mathbf{B}$, all view traceability mappings computed by vExe are directed from $\mathbf{B}$ to $\mathbf{A}$.

A view synchronizer is called *well-behaved (wb)* if the pair $(\mathsf{get}, \mathsf{put})$ satisfies the laws IdlGet IdlPut, and PutGet specified in Fig. 20. A wb synchronizer is called *very well-behaved* if the laws GetGet and PutPut hold as well.

Exercise 7. Prove that in the discrete setting (mappings are just pairs of models), a (very) wb view synchronizer becomes a (very) wb lens [2].

Exercise 8. Let $\lambda_1\colon \mathbf{A} \to \mathbf{B}$, $\lambda_2\colon \mathbf{B} \to \mathbf{C}$ be view synchronizers defined in Sect. 5.2. Define a view synchronizer $\lambda_1;\lambda_2\colon \mathbf{A} \to \mathbf{C}$ and prove that it is (very) well-behaved as soon as the components λ_i are such.
Hint: Define $\mathbf{V}^{\lambda} \stackrel{\text{def}}{=} \big\{(v_1, v_2)_A\colon\ v_1 = \mathsf{vExe}^{\lambda_1}(A), v_2 = \mathsf{vExe}^{\lambda_2}(v_1.\partial_{\mathsf{s}}), A \in \mathbf{A}_0\big\}$ and $\mathsf{vExe}^{\lambda}(A) \stackrel{\text{def}}{=} (v_1, v_2)_A$.

6 Complex Update Propagation: Managing Heterogeneity

In this section we consider scenarios in which the operation of update propagation is assembled from simpler propagation blocks.

6.1 Synchronization of Heterogeneous Models

Suppose that models to be synchronized are instances of different metamodels, for example, we need to keep in sync a class diagram and a sequence diagram. If one of the models is updated, say, a method in the class diagram is renamed, we need to update the sequence diagram and rename messages calling for the renamed method. Thus, we need to propagate updates across a match between heterogeneous (non-similar) models.

We will approach this problem by adapting constructions developed in Sect. 4 for homogeneous replication. Surprisingly, a precise realization of this idea is not too complicated. We will first find "the right" constructs using the metamodels, and then proceed with algebras over spaces models like in the previous section.

Matching. Discussion in Appendix D shows that heterogeneous model matching is based on metamodel matching via a span $\boldsymbol{o} = \boldsymbol{\mathcal{A}} \stackrel{\boldsymbol{v}}{\leftarrow} \boldsymbol{\mathcal{O}} \stackrel{\boldsymbol{w}}{\rightarrow} \boldsymbol{\mathcal{B}}$ in the space of metamodels, where $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ are metamodels of models to be synchronized, $\boldsymbol{\mathcal{O}}$ is a metamodel specifying their overlap, and mappings $\boldsymbol{v}$,$\boldsymbol{w}$ are view definitions that make $\boldsymbol{\mathcal{O}}$ a common view to $\boldsymbol{\mathcal{A}}$, $\boldsymbol{\mathcal{B}}$. Recall that each metamodel $\boldsymbol{\mathcal{M}}$ determines a 1.5-sorted category $\mathbb{M}\mathbf{od}(\boldsymbol{\mathcal{M}})$ whose objects are $\boldsymbol{\mathcal{M}}$-instances (models),

vertical arrows are their updates and horizontal arrows are matches (Sect. 4.1). To simplify notation, we will use the following abbreviations. For a given span $\boldsymbol{o} = \boldsymbol{\mathcal{A}} \xleftarrow{\boldsymbol{v}} \boldsymbol{\mathcal{O}} \xrightarrow{\boldsymbol{w}} \boldsymbol{\mathcal{B}}$, bold letters $\mathbf{A}$, $\mathbf{B}$ denote the *vertical* categories (of updates) in $\mathbb{M}\mathbf{od}(\boldsymbol{\mathcal{A}})$ and $\mathbb{M}\mathbf{od}(\boldsymbol{\mathcal{B}})$ resp; bold letter $\mathbf{O}$ denotes the *horizontal* graph (of matches) in $\mathbb{M}\mathbf{od}(\boldsymbol{\mathcal{O}})$.

We assume that the metamodel span is consistent, that is, there are no conflicts between the metamodels.

Definition 9. A *heterogeneous match* of type $\boldsymbol{o}$ is a triple $h = (A, m, B)$ with $A \in \mathbf{A}_0$, $B \in \mathbf{B}_0$, and $m\colon A{\restriction_{\boldsymbol{v}}} \to B{\restriction_{\boldsymbol{w}}}$ a match between the corresponding projections in graph $\mathbf{O}$. Match h is called *consistent* if match m is such.

Given a metamodel span $\boldsymbol{o}$, we will denote heterogeneous matches of type $\boldsymbol{o}$ by arrows $A \xrightarrow{h:\boldsymbol{o}} B$ or $h_{\boldsymbol{o}}\colon A \to B$. The typing discipline then implies that models A and B are instances of metamodels $\boldsymbol{\mathcal{A}} = \partial_{\mathsf{s}}\boldsymbol{o}$ and $\boldsymbol{\mathcal{B}} = \partial_{\mathsf{t}}\boldsymbol{o}$ resp.

Propagation. Suppose we are given a matched heterogeneous pair of models $h_{\boldsymbol{o}}\colon A \to B$. If one of the models, say A, is updated and consistency between models gets worse, we may want to propagate update $a\colon A \to A'$ to model B and restore consistency as much as possible. Thus, we need to compute an update $b\colon B \to B'$ along with an updated match $h'_{\boldsymbol{o}}\colon A' \to B'$ of the same type $\boldsymbol{o}$.

If both legs of the span $\boldsymbol{o}$ are maintainable views, and the replication space $\mathbb{M}\mathbf{od}(\boldsymbol{\mathcal{O}})$ is equipped with synchronization, a reasonable idea would be to compose update propagation from A to B from the blocks provided by synchronization mechanisms of $\boldsymbol{v}$, $\boldsymbol{\mathcal{O}}$, and $\boldsymbol{w}$. That is, having lenses $\lambda_{\boldsymbol{v}}$ and $\lambda_{\boldsymbol{w}}$, and a homogeneous replica synchronizer δ over $\mathbb{M}\mathbf{od}(\mathcal{O})$, we may try to build a heterogeneous replica synchronizer spanning model spaces $\mathbf{A}$ and $\mathbf{B}$. The rest of the section is devoted to a precise realization of this idea.

After metamodels have helped us to figure out the right concepts, we may forget about them and work within model spaces only.

Definition 10. A *triple lane* $\mathbf{t}$ is a pair of view lanes $(\mathbf{v}^l, \mathbf{v}^r)$ referred to as the *left* and the *right* lanes, with a replica lane in-between them:

$$\mathbf{A} \xrightarrow{\mathbf{v}^l} \mathbf{O} \xrightarrow{\mathbf{r}} \mathbf{O} \xleftarrow{\mathbf{v}^r} \mathbf{B}.$$

Categories $\mathbf{A}$, $\mathbf{B}$ are called the *ends* of the triple lane and category $\mathbf{O}$ is the *overlap*.

A *triple synchronizer* τ over a triple lane $\mathbf{t}$ is a pair of view synchronizers for the pair of view lanes and a diagonal replica synchronizer for the replica lane: $\tau = (\lambda^l, \delta, \lambda^r)$ with $\mathbf{A} \xrightarrow{\lambda^l} \mathbf{O} \xrightarrow{\delta} \mathbf{O} \xleftarrow{\lambda^r} \mathbf{B}$.

A triple synchronizer is called *(very) well-behaved* if all its three components are such.

Theorem 2. *Any triple synchronizer $\tau = (\lambda^l, \delta, \lambda^r)$ gives rise to a diagonal replica synchronizer Δ_τ. Moreover, the latter is (very) well-behaved as soon as all three components are such.*

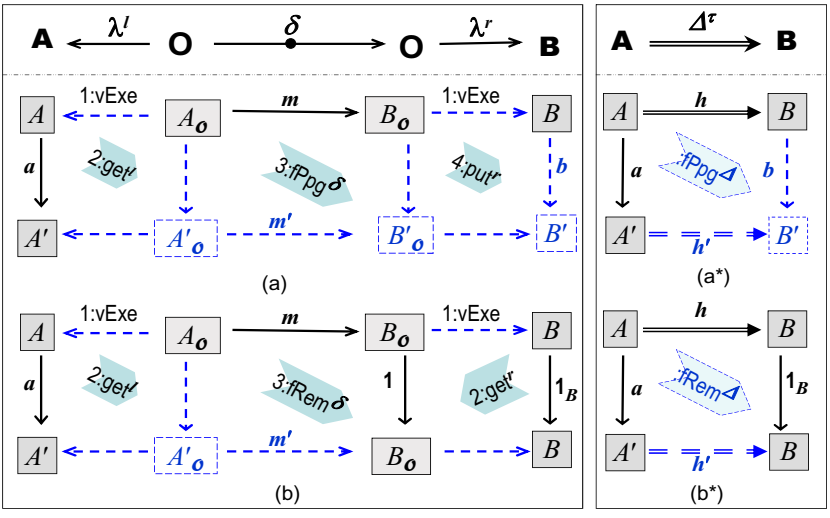


Fig. 22. Heterogeneous update propagation

The principle idea of the proof is easy and well explained by Fig. 22.

The binary replica lane for Δ_τ is formed by the ends of **A** and **B** of the triple lane, and with the class of matches formed by heterogeneous matches (A, m, B) as described in Definition 6.1. The subclass of consistent matches is also described in Definition 6.1.

The four operations of diagonal update propagation specified in Fig. 15 are defined by tiling the corresponding operations of the three component synchronizers: Fig. 22 shows this for forward propagation (a) and rematching (b). Applications of the operations are numbered, and concurrent applications have the same number. Algebraically, diagrams (a) and (b) specify terms that can be abbreviated by diagrams (a*) and (b*). It is exactly similar to definitions by equality in the ordinary algebra: when we write, say, $\Delta(x, y) \stackrel{\text{def}}{=} ax * (b_1 x + b_2 y) * cy$ with x, y variables and a, b, c fixed coefficients, expression $\Delta(x, y)$ can be considered as an abbreviation for the term on the right-hand side of equality symbol. Backward propagation and rematching are defined in exactly the same way but in the opposite direction.

Finally, we need to check that composed operations in diagrams (a*,b*) and their backward analogs satisfy the laws specified in Fig. 16. With tiling notation, this check is straightforward. Ancient Indian mathematicians used to prove their results by drawing a picture and saying "Look!". The reader is encouraged to follow this way and appreciate the benefits of diagram algebra. □

Similarly to unidirectional heterogeneous update propagation, heterogeneous bi-directional operation can be built from lenses and bi-directional synchronization over the overlap as suggested by Fig. 23 (where $\delta^{\Leftrightarrow}$ denotes the operation of homogeneous bi-directional update propagation). A special case of this construction for synchronizing data presented by trees was described in [3].

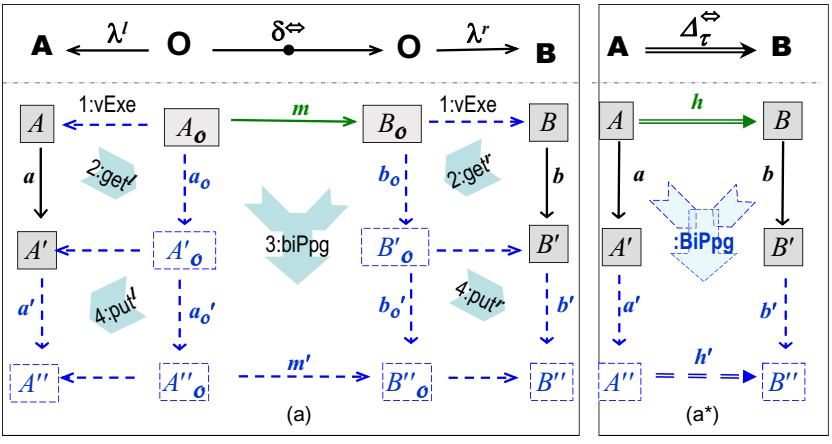


Fig. 23. Heterogeneous bi-directional update propagation

Exercise 9 (*). Define an algebra for modeling synchronization of *materialized* views, for which view data are managed independently, and inconsistency with the source is possible (though undesirable). *Hint*: The possibility of inconsistency makes this case somewhat similar to replication (Sect. 4) and distinct from ordinary views (Sect. 5.2).

6.2 Synchronization with Evolving Metamodels: A Sketch

First we note that typing can be considered as a specific kind of match. Then model adaptation to metamodel evolution can be described as backward diagonal propagation as shown by Fig. 24 (in which superscript ϵ stands for "evolution"). Arrow u encodes an ordinary (update) span in the space of metamodels. Arrow a is a span whose head is an instance of u's head, and the legs are heterogeneous model mappings over u's legs as described in Sect. D.1.

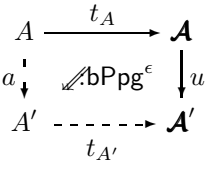


Fig. 24. Typing as matching

Now consider a heterogeneous pair of replicas $A{:}\boldsymbol{\mathcal{A}}$ and $B{:}\boldsymbol{\mathcal{B}}$, and suppose that metamodels may change. A typical scenario is shown in Fig. 25(a). The upper face of the cube specifies a heterogeneous match defined in Sect. D.2. Suppose that metamodel $\boldsymbol{\mathcal{A}}$ is updated with $u{:}\,\boldsymbol{\mathcal{A}} \to \boldsymbol{\mathcal{A}}'$. This update can be propagated in two directions.

In the first one, update u is propagated over the left face of the cube and results in update $a{:}\,A \to A'$ adapting model A to the change. In the second direction, update u is first propagated to metamodel $\boldsymbol{\mathcal{B}}$ along the match $\boldsymbol{o}$ by the ordinary replica synchronization mechanisms (Sect. 5) but now working with the metamodels rather than models. This gives us the back face of the cube and update $v{:}\,\boldsymbol{\mathcal{B}} \to \boldsymbol{\mathcal{B}}'$ of the right metamodel. The latter is then propagated to model B by the model adaptation mechanism now applied to the right face of the cube.

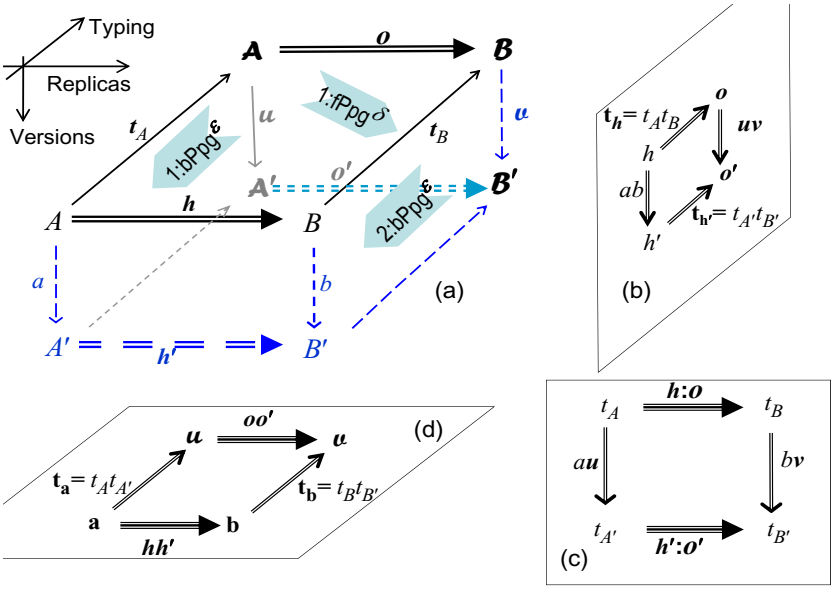


Fig. 25. 3D-synchronization with evolving metamodels

In this way we get two parallel updates a and b at the ends of match h. Having the metamodel span $\boldsymbol{o}' = (\boldsymbol{\mathcal{A}}' \overset{\boldsymbol{v}'}{\leftarrow} \boldsymbol{\mathcal{O}}' \overset{\boldsymbol{w}'}{\rightarrow} \boldsymbol{\mathcal{B}}')$ at the back face, we may project models A' and B' to their common overlap space $\mathbb{M}\mathbf{od}(\boldsymbol{\mathcal{O}}')$ thus arriving at models $A'_{\mathcal{O}} = \boldsymbol{v}'\restriction_{A'}$ and $B'_{\mathcal{O}} = \boldsymbol{w}'\restriction_{B'}$. Having match $m\colon A_{\mathcal{O}} \to B_{\mathcal{O}}$ (occurring into h) and all other information provided by the cube, we may derive a match $m'\colon A'_{\mathcal{O}} \to B'_{\mathcal{O}}$ by applying the corresponding operation to nodes and arrows of the cube (in its de-abbreviated form with all models and model mappings explicated). This would be a typically categorical exercise in diagram chasing. A theoretical obstacle to be watched is that categories involved must be closed under the required operations. Practically, it means that the required operations have to be implemented.

Thus, synchronization scenarios with evolving metamodels are deployed within a *three-sorted* graph with three sorts of arrows: vertical (updates), horizontally frontal (matches) and horizontally "deep" (typing). Since updates and types are composable, we actually have a *2.5-sorted* category. If heterogeneous match composition is also defined, we have a thin *triple* category. This pattern could be probably extended for other types of relationships between models. Hence, general synchronization scenarios are mutli-dimensional and are deployed within n-sorted graphs and categories. *Multi-dimensional category theory (mdCT)* appears to be an adequate mathematical framework for multi-dimensional synchronization.[6]

[6] Md-category seems to be a new term. The term *higher-dimensional* categories is already in use and refers to md-categories with weaker compositional laws: unitality and associativity of composition hold up to canonic isomorphisms [31]. In fact, hdCT is a different discipline, and mdCT is a proper, and very simple, sub-theory of hdCT.

2D-projections. To manage the complexity of 3D-synchronization, it is useful to apply a classic idea of descriptive geometry and study 2D-projections of the 3D-whole. We can realize the idea by arrow encapsulation, that is, by treating arrows of some sort as objects (nodes) and faces between those arrows as morphisms (complex arrows). There are three ways of applying this procedure corresponding to the three ways of viewing the cube (see the frame of reference in the left-upper corner).

Viewing the cube along the axis of Replicas means that we consider match arrows as nodes, the top and bottom faces as "deep" arrows, and the front and back faces as vertical arrows. In this view, the cube becomes a tile shown in diagram (b). If we treat typing mappings as specific matches, these tiles become similar to replica synchronization tiles from Sect. 4

In the view along the Typing axis, typing mappings are nodes, the top and bottom faces are horizontal arrows, and the left and right faces are vertical arrows (but of type different from vertical arrows of the Replicas-view). The result is shown in diagram (c). These tiles are similar to heterogeneous replication considered above but with evolving metamodels.

Finally, in the Versions view, updates are nodes, the front and back faces are new horizontal arrows, and the left and right faces are new deep arrows as shown in diagram (d). Such tiles can be seen as structures for specifying "dynamic typing", in which typing arrows are actually couples of original and updated typing mappings.

Tiles of each of the three sorts can be repeated in the respective directions and we come to three two-sorted graphs $\mathbb{G}_{\mathsf{x}}$ with $\mathsf{x} = R, T, V$ for the Replicas, Typing, Version axes. Each of the graphs is a universe for its own synchronization scenarios with different contexts. Yet there may be many similarities in the algebras of operations, and there may be core algebraic structures common to all three views. We leave this for future work.

7 Relation to Other Work, Brief Discussions, Future Work

The paper is a part of an ongoing research project on model synchronization with the Generative Software Development Lab at the University of Waterloo The project started with the GTTSE'07 paper [32] by Antkiewicz and Czarnecki, which outlined a broad landscape of heterogeneous synchronization, provided a range of examples, and introduced a notation that can be seen as a precursor of synchronization tiles. The project has been further developed in [33,11,34], and in several papers currently in progress. The present paper aims to specify a basic mathematical framework for the project, and to offer a handy yet precise notation.

Of course, this is only the short prehistory of the paper. Synchronization spans a wide range of specification problems, and the present paper (in its attempt to set a sufficiently general framework) inevitably intersects with many ideas

and approaches, and builds on them. These "pre-histories" and intersections are briefly reviewed below without any aspiration to be complete. Directions for future work are also discussed.

7.1 Abstract MMt or *Model-at-a-Time* Programming

Synchronization via algebra. Analysis of synchronization problems in abstract algebraic setting is a long-standing tradition in the literature on databases and software engineering. It can be traced back to early work on the view update problem, particularly to seminal papers by Bancilhon and Spyratos [35], Dayal and Bernstein [30] and Gottlob *et al* [36]. This algebraic style was continued by Meertens in [37] in the context of constraint maintenance, and more recently was further elaborated in the work of the Harmony group on bi-directional programming languages and data synchronization [2,3,38,39,40]. An adaptation of the approach for bi-directional model transformations was developed by Stevens [6,41] and Xiong *et al* [4,42]; an analysis of the corresponding algebraic theories can be found in [28]. Paper [32] mentioned above, and an elegant relational model of bi-directional data transformations [43] by Oliveira are also within this algebraic trend.

Two features characterize the framework: (A) model mappings are not considered or implicit; and (B) metamodels (and their mappings) are either ignored or only considered extensionally —- a metamodel defines its class of instances and may be forgotten afterwards (e.g., see [3]).

Feature (A) makes the framework discrete and subject to the critique in Section 2. Feature (B) significantly simplifies technicalities but hides semantics of model translation and makes it difficult to manage heterogeneity in a controlled way.[7] The abstract MMt part of the present paper also does not include metamodels. However, the latter are central for the concrete MMt part that motivates and explains several important constructs in the abstract part.

Generic MMt. A broad vision of the model-at-a-time approach to the database metadata management was formulated by Bernstein *et al* in [44,1]. They coined the term *generic model management*, stressed the primary importance of mappings, and described several major operations with models and mappings necessary to establish a core MMt framework. This work originated a research direction surveyed in [45]. Interestingly, synchronization operations are not included in the core framework as scoped by Bernstein *et al*.

Although mappings are first-class citizens in generic MMt, a typical algebraic setting is discrete: the universe in which operations act is a set (of models and mappings) rather than a graph; the same setting is used in Manifesto [46] by Brunet *et al*. Not surprisingly, neither diagram algebra nor category

[7] Dayal and Bernstein's work [30] is a notable exception. It does use update, traceability and typing links (and, in fact, is remarkably categorical in its approach to the problem). However, these links are not organized into mappings (not to mention more advanced arrow encapsulation techniques), and technicalities become hardly manageable. The categorical potential of the paper remained undiscovered.

theory are employed in generic MMt so far; few exceptions like [47] by Alagic and Bernstein, and [48] by the present author remain episodes without follow-up. The purely extensional treatment of data schemas (feature (B) of synchronization frameworks above) is also typical for generic MMt [49]; papers [47,48] are again exceptions.

Tiles and tiling. *Tile systems* were developed by Ugo Montanari *et al* (see [50] and references therein) as a general algebraic framework for modular description of concurrent systems. The tiles represent modules and can be thought of as computations (or conditional rewriting rules) with side effects. The two horizontal arrows of a tile are the initial and the final states of the module, and the two vertical arrows are the trigger and the effect. This interpretation works for our tiles: modules are connected pairs of models, matches are their states, the input update is a trigger and the output one is the effect. However, there are important distinctions between the two tile frameworks. For the brief discussion below, we will refer to them as to *c-tiles* and *s-tiles*, with *c* standing for *concurrency* and *s* for synchronization.

(a) C-tiles have an interior in the sense that different c-tiles may have the same four-arrow boundary whereas our s-tiles are merely quadruples of arrows (in the categorical jargon, they are *thin*).

(b) Montanari *et al* only deal with operations on tiles as integral entities (*tiling-in-the-large*), and consider their vertical, horizontal and parallel composition. In contrast, we have been looking inside tiles and considered algebraic operations that produce tiles from arrows (*tiling-in-the-small*). We have also considered vertical composition-in-the-large in our XyzXyz laws, and horizontal composition in Exercise 6 on p.126.

(c) Three composition operations over c-tiles assume they are homogeneous units, and so we have *homogeneous* tiling. In contrast, our complex scenarios in Sect. 6 present *heterogeneous* tiling: a big tile is composed from smaller tiles of different types.

A perfectly adequate mathematical framework for homogeneous tiling is *double categories* [25], or *two-sorted categories* (Appendix B) for thin tiles; their s-interpretation is described in [33]. Heterogeneous tiling requires more refined algebraic means and a real diagram algebra. A general formal definition of a diagram operation appears in [51] and is specified in detail in [52]; in the present paper it is formulated in a slightly different but equivalent way. Parsing terms composed of diagram operations is discussed in [52, Appendix A].

A few historical remarks. Elements of the tile language in the context of model synchronization can be found in Antkiewicz and Czarnecki [32], and even earlier in Lämmel [53]; my paper [28] also deals with s-tiles but in the discrete setting. Operations of update propagation and conflict resolution are considered in [32] but without any equational laws. The language of s-tiles is explicitly introduced in Diskin *et al* [33] with a focus on 2D-composition and double categories. A general framework for specifying synchronization procedures via tile algebra in this paper is novel.

The 2-arrow structure. Introducing a partial order on mappings, and then ordering matches and updates, is important for model synchronization (see p. 116) and should be a part of the tile language. The issue is omitted in the paper and left for future work; some preliminary remarks are presented below.

By the principles of arrow thinking, ordering should be modeled by arrows, and we thus come to arrows between arrows or *2-arrows*. If mappings are spans, 2-arrows are ordinary arrows between their heads, but the entire structure becomes a 2-category and hence a much richer structure than an ordinary category. Another approach suggested by an anonymous referee is to work with so called *allegories* [54] rather than categories, in which morphisms are to be thought of as binary relations rather than functions. However, an important feature of any set of matching links — its structure being similar to the structure of models — is lost if mappings are simply morphisms in an allegory. Another (arguable) advantage of the span model of mappings is that it is technically easier to work with 2-categories of spans than with allegories.

Parallel updates. This synchronization scenario is very important yet omitted in the paper and left for the future work. It is a challenging problem, whose algebraic treatment needs a more elaborated framework than simple algebraic models we used. An initial attempt and some results can be found in [42].

Lenses, view synchronization and categories. The Harmony group's paper [55] was seminal. It presented a basic algebraic framework in a very transparent way; and coined several vigorous names: *get* and *put* for the two main operations, GetPut, PutGet, PutPut for equational laws imposed on these operations, and *lens* for the resulting "bi-directional" algebra. In fact, the paper set up a pattern for algebraic models of update propagation.

The basic lens framework is enriched with update mappings in [11]. Algebras introduced in [11] operate on both models and update mappings, and are called *u-lenses* with 'u' standing for updates. Earlier, a similar framework was developed by Johnson and Rosebrugh [26]. For them, updates are also arrows, a model space is a category, and a view is a functor. However, they work in the concrete rather than abstract MMt setting, and focus on conditions ensuring uniqueness of update policies. As discussed in Sect. 4.2, this setting may be very restrictive in practice.

View synchronizers of the present paper can be seen as *ut-lenses* since they operate on two types of mappings: updates and traceability. Moreover, given a view definition language with well-behaved operations of update propagation defined for any view mapping, both tile systems, of all get-tiles and of all put-tiles, give rise to two-sorted categories, say, $\mathbb{G}\boldsymbol{et}$ and $\mathbb{P}\boldsymbol{ut}$ (see Fig. 20 and Exercise 6 on p.127). In addition, the PutGet law entails inclusion $\mathbb{P}\boldsymbol{ut} \subset \mathbb{G}\boldsymbol{et}$. Proving these results is not difficult and will appear elsewhere.

Multi-dimensional synchronization. The ideas of constructing 3D-tiles (synchronization cube on p. 132), and more generally of the multi-dimensional nature of synchronization problems, seem to be new. The paper only presents a vision (Sect. 6.2), and even the initial steps are still waiting for a precise specification.

With dynamic interpretation of horizontal arrows as transformations (rather than structural mappings), 2D-projections of the synchronization cube can be seen as *coupled transformations* considered in [53], and have probably been studied by different communities in different contexts, e.g., [56]. If vertical arrows (updates) are interpreted dynamically, then the front and back faces of the cube become close to triple graph transformations [57] (with the third graph hidden in the match). Stating precise relationships is a future work.

7.2 Concrete Model Management

Inside models: constraints as diagrams predicates. For a rich software model, specifying its abstract syntax "tree" as a mathematical object is not as easy as it may seem. One of the challenges is how to specify and manage constraints, which populate model graphs with non-instantiable elements. In the paper we have only considered very simple constraints declared for a single arrow (multiplicities). However, there are other practically important constraints involving several arrows, e.g., invertibility of two mappings going in the opposite directions, uniqueness of identification provided by a family of mappings with a common source (a key), and many other conditions that constraint languages (like OCL) allow us to specify.

A general approach to the problem is to specify such constraints as *diagram predicates* [51] and treat models as graphs with diagram predicates, *dp-graphs*. A principal distinction of this approach from the attributed typed graphs (ATGs) [58] is that a constraint is an independent object referring to respective nodes and edges rather than an attribute of a node or an edge. Theoretical advantages of the approach are its universality and proximity to an established framework of the so called *sketches* developed in categorical logic (see [16] for details). The approach was shown to be useful in schema integration [59], conceptual modeling [60], and fixing known problems with UML associations [61]. An accurate algebraic model of metamodeling with diagrammatic constraints is an important direction for future work.

Homogeneous model mappings and deltas. Specifying (symmetric) deltas is a known issue, e.g., [9,10,62,63]. A major challenge is how to formally specify model *changes*: *modifications*, if we interpret deltas as updates, or *conflicts*, if deltas are matches. A well-known idea is to treat a modification of an element as its deletion followed by insertion; but it is a simplistic treatment. The approach developed in the paper (for our OSV-models) is more adequate and still simple but is not straightforward. First, value-preserving model mappings are defined; then changes are specified by spans built from two value-preserving mappings but having empty slots. This treatment of changes seems correlating with ATG transformations but a precise comparison needs some technical work to be done. Generalization of the idea for more practically interesting (and hence more complex) models than simple OSV-models is important future work.

Heterogeneous model mappings and deltas. Precise specification of operations on heterogeneous models and model mappings is a rarity in the literature because of semantic and technical difficulties. It is managed in the paper by specifying heterogeneous models as chains of graphs and graph mappings; model mappings then appear as multi-layer commutative diagrams. The idea seems to be more or less evident but I am not aware of its realization in the literature.

Despite their frightening appearance, universes of multi-layer complex objects and mappings are well-known in CT under the name of *arrow categories*. They are well-studied and behave very well. Unfortunately, constraints may be an obstacle: while any model is a chain of graph mappings, not any such chain is a model because it may violate the constraints. It implies that the universe of models may be *not* closed wrt. some operations, e.g., merging (colimit) [34]. How graph-based constraints declared in a metamodel affect the properties of the corresponding universes of models is a big issue studied in categorical logic. Its adaptation to metamodeling with diagram predicates (as constraints) [16] is important future work.

Model translation (MT) and fibrations. The algebraic model of MT proposed in the paper is generic and formulated for any metamodel language, including an associated query language. In this model, MT is treated as a view computation, and is entirely determined by the corresponding metamodel mapping considered as a view definition. The idea was first described in [48]; the description in the present paper is more accurate and detailed. It culminates in the statement that the view mechanism (for monotonic queries) makes the functor projecting heterogeneous models and mappings to their metamodel components a split *fibration* — a construct well known and studied in CT.

Fibrational formulation can be seen as dual to the familiar *functorial semantics*: a model is a functor from the metamodel (theory) to some semantic category, e.g., of sets and relations. Functorial semantics is quite popular in the Algebraic Specification community [64], and is basic for the categorical approach to the view update problem developed in [26], but it may seem foreign for a model transformation engineer accustomed to work with metamodeling patterns. The latter assume that a model is given by a (typing) mappings *to* the metamodel rather than *from* it. Fibrations fit perfectly in this framework, but offer much more. Practical modeling situations often comprise instances at several levels, say, objects, classes, and the metamodel for the latter (e.g., a simple sequence diagram is a three-level structure of this kind [65]). Specification of multilevel modeling is quite manageable with fibrations: composition of fibrations is again a fibration (this is a well-known result). In contrast, functorial semantics becomes hard to manage when we consider more than one pair (theory, model).

8 Conclusion

Building theoretical foundations for model synchronization is a challenging problem. Among the factors contributing to its complexity are heterogeneity of

models to be synchronized, the multitude and heterogeneity of their relationships, and interactions between different dimensions of synchronization. The paper aims to show that algebraic models based on diagram operations can be an effective means to manage the complexity of the problem.

Two lines of approaches and results are presented. The first one is *abstract*: models and model mappings are treated as indivisible (black-box) nodes and arrows, on which synchronization procedures operate. The machinery used is algebra of tile operations and tiling as term substitution. The abstract line culminates in Sect. 6, which shows how complex synchronizers can be assembled by tiling together simple components. The second line is *concrete*: it provides algebraic models for (white-box) complex structures underlying models and model mappings. The machinery is essentially categorical: arrow categories (for heterogeneous models and their mappings) and fibrations (for the view mechanism). Tile algebra is applicable here as well.

The tile framework offers a handy notation with formal semantics, and a toolbox of constructs amenable to algebraic manipulations and hence to automated computer processing. This benefit package may be very appealing for a software engineer.

Synchronization scenarios considered in the paper are deployed on 2D-planes of a 3D-space populated by models and model mappings (and a 3D-scenario with evolving metamodels is sketched in Sect. 6.2). The three dimensions correspond to the three kinds of intermodel relationships (and mappings) that were considered: replication (matches), versioning (updates), metamodeling (typing). Other kinds of relationships can give rise to new dimensions of the space and synchronization procedures spanning it. Handy yet precise tile notation and the corresponding algebraic framework can be an invaluable tool for multi-dimensional synchronization.

Acknowledgements. I'm grateful to the organizers of GTTSE'09 for the invitation to lecture at the school and for the chance to enjoy its unique atmosphere — stimulating, challenging, and warm. I'd like to thank the same team for organizing qualified and friendly reviewing, and their enormous editor's patience. I'm indebted to the anonymous referees for valuable feedback and criticism, which greatly improved the structure and presentation of the paper. Special thanks to the referee who also provided detailed technical comments and suggestions.

Different parts of the paper were presented at our seminars at the University of Waterloo, and I'm grateful to Michał Antkiewicz, Krzysztof Czarnecki and Yingfei Xiong for valuable feedback and constructive criticism. Several discussions of the paper's general ideas and technical fragments with Tom Maibaum were thought-provoking and invigorating. In addition, Tom read the entire manuscript and provided many helpful remarks. Finally, thanks to GTTSE'09 students and participants for their attention, stimulating questions, and discussions that supported my enthusiasm about the practical use of abstract diagrams.

Financial support was provided by the Ontario Research Fund.

References

1. Bernstein, P.: Applying model management to classical metadata problems. In: Proc. CIDR 2003, pp. 209–220 (2003)
2. Foster, J.N., Greenwald, M., Moore, J., Pierce, B., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. 29(3) (2007)
3. Foster, J.N., Greenwald, M., Kirkegaard, C., Pierce, B., Schmitt, A.: Exploiting schemas in data synchronization. J. Comput. Syst. Sci. 73(4), 669–689 (2007)
4. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: ASE, pp. 164–173 (2007)
5. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: ICFP, pp. 47–58 (2007)
6. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
7. Czarnecki, K., Foster, J., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.: Bidirectional transformations: A cross-discipline perspective. [66], 260–283
8. Xing, Z., Stroulia, E.: Umldiff: an algorithm for object-oriented design differencing. In: Redmiles, D., Ellman, T., Zisman, A. (eds.) ASE, pp. 54–65. ACM, New York (2005)
9. Lin, Y., Gray, J., Jouault, F.: DSMDiff: A Differentiation Tool for Domain-Specific Models. European J. of Information Systems 16, 349–361 (2007)
10. Treude, C., Berlik, S., Wenzel, S., Kelter, U.: Difference computation of large models. In: ESEC/SIFSOFT FSE, pp. 295–304 (2007)
11. Diskin, Z., Xiong, Y., Czarnecki, K.: From state- to delta-based bidirectional model transformations. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 61–76. Springer, Heidelberg (2010)
12. Balzer, R.: Tolerating inconsistency. In: ICSE, pp. 158–165 (1991)
13. Melnik, S., Rahm, E., Bernstein, P.: Developing metadata-intensive applications with Rondo. J. Web Semantics 1, 47–74 (2003)
14. Dyreson, C.E.: A bibliography on uncertainty management in information systems. In: Uncertainty Management in Information Systems, pp. 415–458 (1996)
15. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley, Reading (2004)
16. Diskin, Z., Wolter, U.: A diagrammatic logic for object-oriented visual modeling. Electron. Notes Theor. Comput. Sci. 203(6), 19–41 (2008)
17. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In: ICDE, pp. 117–128. IEEE Computer Society, Los Alamitos (2002)
18. Falleri, J.R., Huchard, M., Lafourcade, M., Nebut, C.: Metamodel matching for automatic model transformation generation. [67], 326–340
19. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. 37(1), 42–81 (2005)
20. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Software Eng. 28(5), 449–462 (2002)
21. Alanen, M., Porres, I.: Difference and union of models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 2–17. Springer, Heidelberg (2003)
22. Ohst, D., Welle, M., Kelter, U.: Differences between versions of uml diagrams. In: ESEC / SIGSOFT FSE, pp. 227–236. ACM, New York (2003)

23. Gadducci, F., Montanari, U.: The tile model. In: Proof, Language, and Interaction, pp. 133–166 (2000)
24. Sabetzadeh, M., Easterbrook, S.: An algebraic framework for merging incomplete and inconsistent views. In: 13th Int. Conference on Requirement Engineering (2005)
25. Kelly, G., Street, R.: Review of the elements of 2-categories. In: Category Seminar, Sydney 1972/73. Lecture Notes in Math., vol. 420, pp. 75–103 (1974)
26. Johnson, M., Rosebrugh, R.: Fibrations and universal view updatability. Theor. Comput. Sci. 388(1-3), 109–129 (2007)
27. Goguen, J.A., Meseguer, J.: Order-sorted algebra i: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theor. Comput. Sci. 105(2), 217–273 (1992)
28. Diskin, Z.: Algebraic models for bidirectional model synchronization. In: [67], pp. 21–36. Springer, Heidelberg (2008)
29. Liefke, H., Davidson, S.: View maintenance for hierarchical semistructured data. In: Kambayashi, Y., Mohania, M., Tjoa, A.M. (eds.) DaWaK 2000. LNCS, vol. 1874, pp. 114–125. Springer, Heidelberg (2000)
30. Dayal, U., Bernstein, P.: On the correct translation of update operations on relational views. TODS 7(3), 381–416 (1982)
31. Leinster, T.: Higher Operads, Higher Categories. Cambridge University Press, Cambridge (2004)
32. Antkiewicz, M., Czarnecki, K.: Design space of heterogeneous synchronization. In: [68], pp. 3–46
33. Diskin, Z., Czarnecki, K., Antkiewicz, M.: Model-versioning-in-the-large: Algebraic foundations and the tile notation. In: ICSE 2009 Workshop on Comparison and Versioning of Software Models, pp. 7–12. ACM, New York (2009), doi: 10.1109/CVSM.2009.5071715, ISBN: 978-1-4244-3714-6
34. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying overlaps of heterogeneous models for global consistency checking. In: First International Workshop on Model-Driven Interoperability, MDI 2010, pp. 42–51. ACM, New York (2010), doi: 10.1145/1866272.1866279, ISBN: 978-1-4503-0292-0
35. Bancilhon, F., Spyratos, N.: Update semantics of relational views. TODS 6(4), 557–575 (1981)
36. Gottlob, G., Paolini, P., Zicari, R.: Properties and update semantics of consistent views. ACM TODS 13(4), 486–524 (1988)
37. Meertens, L.: Designing constraint maintainers for user interaction (1998), `http://www.kestrel.edu/home/people/meertens/`
38. Bohannon, A., Foster, J.N., Pierce, B., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: POPL, pp. 407–419 (2008)
39. Bohannon, A., Pierce, B., Vaughan, J.: Relational lenses: a language for updatable views. In: PODS (2006)
40. Hofmann, M., Pierce, B., Wagner, D.: Symmetric lenses. In: POPL (2011)
41. Stevens, P.: A landscape of bidirectional model transformations. [68], 408–424
42. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Supporting parallel updates with bidirectional model transformations. [66], 213–228
43. Oliveira, J.N.: Transforming data by calculation. [68], 134–195
44. Bernstein, P., Halevy, A., Pottinger, R.: A vision for management of complex models. SIGMOD Record 29(4), 55–63 (2000)
45. Bernstein, P., Melnik, S.: Model management 2.0: manipulating richer mappings. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD 2007, pp. 1–12. ACM Press, New York (2007)

46. Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., Sabetzadeh, M.: A manifesto for model merging. In: GaMMa (2006)
47. Alagic, S., Bernstein, P.: A model theory for generic schema management. In: Ghelli, G., Grahne, G. (eds.) DBPL 2001. LNCS, vol. 2397, p. 228. Springer, Heidelberg (2002)
48. Diskin, Z.: Mathematics of generic specifications for model management. In: Rivero, L., Doorn, J., Ferraggine, V. (eds.) Encyclopedia of Database Technologies and Applications, pp. 351–366. Idea Group, USA (2005)
49. Melnik, S., Bernstein, P., Halevy, A., Rahm, E.: Supporting executable mappings in model management. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD 2005, pp. 167–178. ACM Press, New York (2005)
50. Bruni, R., Meseguer, J., Montanari, U.: Symmetric monoidal and cartesian double categories as a semantic framework for tile logic. Mathematical Structures in Computer Science 12(1), 53–90 (2002)
51. Diskin, Z.: Towards algebraic graph-based model theory for computer science. Bulletin of Symbolic Logic 3, 144–145 (1997)
52. Diskin, Z.: Databases as diagram algebras: Specifying queries and views via the graph-based logic of skethes. Technical Report 9602, Frame Inform Systems, Riga, Latvia (1996), http://www.cs.toronto.edu/~zdiskin/Pubs/TR-9602.pdf
53. Lammel, R.: Coupled software transformation. In: Workshop on Software Evolution and TRanformation (2004)
54. Freyd, P., Scedrov, A.: Categories, Allegories. Elsevier Science Publishers, Amsterdam (1990)
55. Foster, J., Greenwald, M., Moore, J., Pierce, B., Schmitt, A.: Combinators for bidirectional tree transformations: a linguistic approach to the view update problem. In: POPL, pp. 233–246 (2005)
56. Berdaguer, P., Cunha, A., Pacheco, H., Visser, J.: Coupled schema transformation and data conversion for xml and sql. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 290–304. Springer, Heidelberg (2006)
57. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 411–425. Springer, Heidelberg (2008)
58. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph tranformations. Springer, Heidelberg (2006)
59. Cadish, B., Diskin, Z.: Heterogenious view integration via sketches and equations. In: Michalewicz, M., Raś, Z.W. (eds.) ISMIS 1996. LNCS (LNAI), vol. 1079, pp. 603–612. Springer, Heidelberg (1996)
60. Diskin, Z., Kadish, B.: Variable set semantics for keyed generalized sketches: Formal semantics for object identity and abstract syntax for conceptual modeling. Data & Knowledge Engineering 47, 1–59 (2003)
61. Diskin, Z., Easterbrook, S., Dingel, J.: Engineering associations: from models to code and back through semantics. In: Paige, R., Meyer, B. (eds.) TOOLS Europe 2008, LNBIP, vol. 11. Springer, Heidelberg (2008)
62. Cicchetti, A., Ruscio, D.D., Pierantonio, A.: A metamodel independent approach to difference representation. JOT 6 (2007)
63. Abi-Antoun, M., Aldrich, J., Nahas, N.H., Schmerl, B.R., Garlan, D.: Differencing and merging of architectural views. Autom. Softw. Eng. 15(1), 35–74 (2008)
64. Ehrig, H., Große-Rhode, M., Wolter, U.: Application of category theory to the area of algebaric specifications in computer science. Applied Categorical Structures 6, 1–35 (1998)
65. Liang, H., Diskin, Z., Dingel, J., Posse, E.: A general approach for scenario integration. [67], 204–218

66. Paige, R.F. (ed.): ICMT 2009. LNCS, vol. 5563. Springer, Heidelberg (2009)
67. Czarnecki, K., Ober, I., Bruel, J., Uhl, A., Völter, M. (eds.): MODELS 2008. LNCS, vol. 5301. Springer, Heidelberg (2008)
68. Lämmel, R., Visser, J., Saraiva, J. (eds.): GTTSE 2007. LNCS, vol. 5235. Springer, Heidelberg (2008)
69. Barr, M., Wells, C.: Category theory for computing science. Prentice Hall, Englewood Cliffs (1995)
70. Adamek, J., Herrlich, H., Strecker, G.: Abstarct and concrete categories. The joy of cats. TAC Reprints, No.17 (2007)
71. Bruni, R., Gadducci, F.: Some algebraic laws for spans. Electr. Notes Theor. Comput. Sci. 44(3) (2001)
72. Diskin, Z.: Model transformation as view computation: an algebraic approach. Technical Report CSRG-573, the University of Toronto (2008)
73. Manes, E.: Algebraic Theories. Graduate Text in Mathematics. Springer, Heidelberg (1976)
74. Jacobs, B.: Categorical logic and type theory. Elsevier Science Publishers, Amsterdam (1999)

Answers to *-Exercises

Exercise 2 (p. 119) Diagram Fig. 14(c1) says that from a match between replicas a new match can be computed *without* changing the replicas. This situation is typical and is a built-in procedure in many differencing tools. However, it cannot be modeled by an algebraic operation of the arity shown in the figure: to recompute a match a new information is required. That is, we may have a reasonable "binary" operation $(m, X) \bullet\!\!\rightarrow m'$ with the second argument standing for contextual information about replicas, but the "unary" operation $m \bullet\!\!\rightarrow m'$ is not too sensible. In contrast, the operation specified by diagram (c2) is quite reasonable and may be called *rematching*: having one of the replicas updated, we recompute the match based on data in the original match and the update.

Exercise 4(p. 123)
Three reasonable laws the operation should satisfy are specified in Fig. 26. Diagram (a1) states that nothing is done with consistent replicas. Diagram (a2) says that conflict resolution is an idempotent operation. Match m' produced by the operation is not supposed to be necessarily consistent: some of the conflicts embodied in match m may need additional information and user's input, and hence cannot be resolved automatically. Yet everything that could be done automatically is done with the first run of the operation. Diagram (a3) says that resolution is complete in the sense that nothing can be propagated in the tile produced by Res.

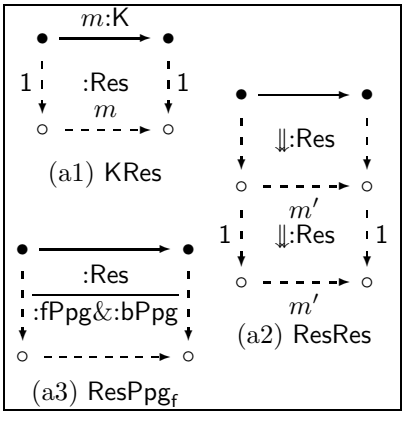


Fig. 26. Laws of conflict resolution

Exercise 9 (p. 131) Synchronization of materialized views can be considered as a particular cases of triple synchronization, in which one view (say, the right one) is identity. Formal definitions are as follows.

A *semi-triple lane* is a pair $\mathbf{st} = (\mathbf{v}, \mathbf{r})$ with $\mathbf{v}$ a view and $\mathbf{r}$ a replica lane coordinated as follows: $\mathbf{A} \xrightarrow{\mathbf{v}} \mathbf{B} \xrightarrow{\mathbf{r}\bullet} \mathbf{B}$. A *semi-triple synchronizer* σ over a semi-triple lane $\mathbf{st}$ is a pair $\sigma = (\lambda, \delta)$ of a view synchronizer λ over the view lane and a diagonal replica synchronizer δ over the replica lane. A semi-triple synchronizer is called *(very) well-behaved* if its two components are such.

The following result is analogous to Theorem 2.

Theorem 3. *Any semi-triple synchronizer $\sigma = (\lambda, \delta)$ gives rise to a diagonal replica synchronizer Δ_σ. Moreover, the latter is (very) well-behaved as soon as its two components are such.*

Appendices
Concrete MMt and Category Theory

Several words about category theory (CT) are in order. CT provides a number of patterns for structure specification and operation. Since models and model mappings are rich structures, and MMt needs to operate them, CT should be of direct relevance for MMt. Of course, this theoretical prerequisite requires practical justification and examples.

Two fundamental categorical ideas are used in the paper.

Encapsulation 1: "To objectify means to mappify". The internal structure of models and model mappings is encapsulated. Models are considered as indivisible objects (points), and mappings as indivisible morphisms (arrows) between them. Mappings of the same type can be sequentially composed and form a category (a graph with associatively composable arrows). Although objects are encapsulated, the categorical language provides sufficient means to recover the internal structure of objects via mappings adjoint to them. For example, a special family of mappings with a common source object makes this object similar to a relation (and its "elements" can be thought of as tuples). Dually, a special family of mappings with a common target object makes it similar to a disjoint union (and its "elements" can be thought of as "either..or" variants). The next section shows how it works.

Encapsulation 2: Arrow categories. Repeatable constructions consisting of several models and mappings are considered as new complex objects or arrows, which can again be encapsulated and so on. In this way we come to categories whose objects (nodes) themselves consist of arrows, while morphisms (arrows) are complex diagrams. For example, a model is, in fact, a typing mapping, and a traceability mapping is a commutative square diagram like the top face of cube Fig. 3(b). Deltas-as-spans denoted by arrows are another simple example. We will build progressively more complex arrow categories in the subsequent sections C and D. Formalization of the sketch presented in Sect. 6.2 requires even more complex arrow encapsulating constructions.

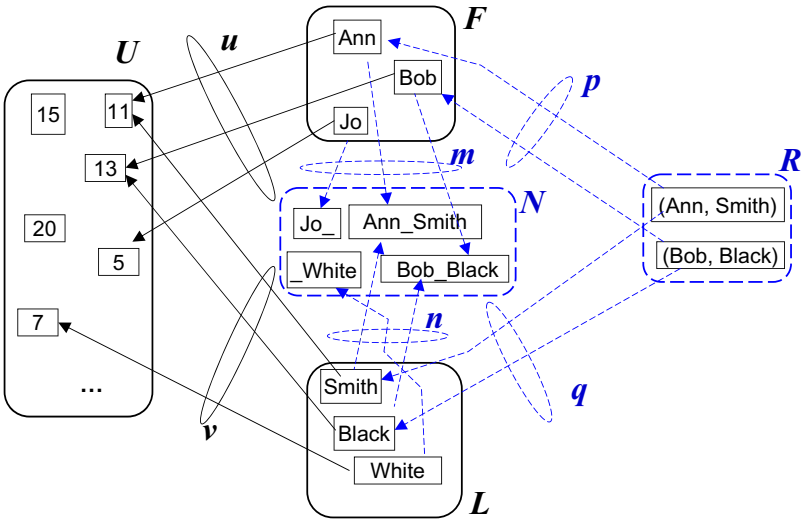


Fig. 27. Matching and merging sets via elements

A Match and Merge as Diagram Operations: Warming Up for Category Theory

This section aims to give a notion of how basic categorical patterns can work in MMt. We will begin with a very simple model of models by considering them as sets of unstructured elements (points), and discuss matching and merging sets. Then we will reformulate the example in abstract terms and come to categories.

A.1 Matching and Merging via Elements

Suppose that our models are sets of strings denoting names, and we have two sets, F of First and L of Last names, of some group of people as shown in Fig. 27. We also assume that for each of the sets, different elements refer to different persons. It does not exclude the situation when an F-name and an L-name refer to the same person, but without additional information, sets F and L are entirely unrelated and disjoint. To match the sets, we map them into some common universe U, say, by assigning to each string the social security number (SSN) of the corresponding person as shown in the left part of the figure. Following UML, we call such assignments *(directed) links* and denote them by arrows (Ann→11, Bob→13 and so on); speaking formally, links are just ordered pairs. Similar (i.e., having the same source and target) links are collected into *mappings*, $u\colon F \Rightarrow U$ and $v\colon L \Rightarrow U$, which are denoted by double-body arrows to distinguish them from link-arrows. We call triple (U, u, v) a *matching cospan* between sets F and L, set U is its *head* and mappings u, v are the *legs*.

Now we may form set $R = \{(x, y)\colon\ u(x) = v(y)\}$ consisting of those pairs of names, which are mapped to the same SSN. This set is equipped with two

projection mappings $p\colon R \Rightarrow F$, $q\colon R \Rightarrow L$ giving the components of the pairs. The way we built R implies that sequential compositions of mappings $p;u$ and $q;v$ are equal: $(x,y).p.u = (x,y).q.v$ for any element $(x,y) \in R$. The triple (R,p,q) is called a *correspondence span* or *matching span* between sets F and L; set R is its *head* and mappings p,q are the *legs*. To show that the components of the span are *derived* from mappings u,v, they are denoted by dashed lines (blue with a color display).

Each pair (x,y) in the head R of the span says that actually elements $x.p \in F$ and $y.q \in L$ refer to the same object of the real world (at least, to the same SSN). Hence, we may be interested in merging sets F and L without duplication of information, that is, by gluing together the first and last names of the same person. Set N of names in the middle of Fig. 27 presents the result. It is formed by first taking disjoint union of sets F and L, and then gluing together those elements, which are declared to be the same by the span. For example, we join Ann and Smith since there is a pair (Ann, Smith) in set R. Since there are two elements in R, set N has four (rather than six) elements. Note also mappings $m\colon F \Rightarrow N$ and $n\colon L \Rightarrow N$ embedding the original sets into the merge.

How joined names are formed is a matter of taste: Ann_Smith, or Ann*Smith or AnnSmith will all work to show that Ann and Smith are two different representations of the same object. In fact, all work is done by inclusion mappings m and n that map Ann and Smith to the same element in N. Similarly, the concrete nature of elements in set R does not matter: it is mappings p,q that do the job and specify that elements of R are pairs. Hence, strictly speaking, sets R and N may be defined up to isomorphism: the internal structure of their elements is not important.

Since the internal structure of elements in sets R and N is not important, it is tempting to try to rewrite the entire construction in terms of sets and mappings only, without elements at all. Such a *pointfree* rewriting, apart of satisfying purely intellectual curiosity, would be practically useful too. If we were able to specify object matching and merging only based on mappings between objects without use of their internal structure, we would have generic patterns of match and merge working similarly for such diverse objects as sets, graphs, typed attributed graphs and so on. The benefits are essential and justify some technical work to be done.

A.2 Matching and Merging via Arrows

Matching. Figure 28(a) presents a more abstract view of our matching construction. Nodes denote sets, and arrows are mappings (functions) between them. Double-frames of nodes and double-bodies of arrows remind us that they have *extension*, i.e., consist of elements (points and links respectively).

Labels in square brackets denote *diagram predicates*, that is, properties of arrow diagrams on which these labels are "hung". Label [=] is assigned to the entire square diagram and declares its commutativity, that is, the property $p;u = q;v$ (i.e., in terms of elements, $r.p.u = r.q.v$ for any $r \in R$). Label [key] is assigned to the arrow span (p,q) and declares the following property: for any $r_1, r_2 \in R$,

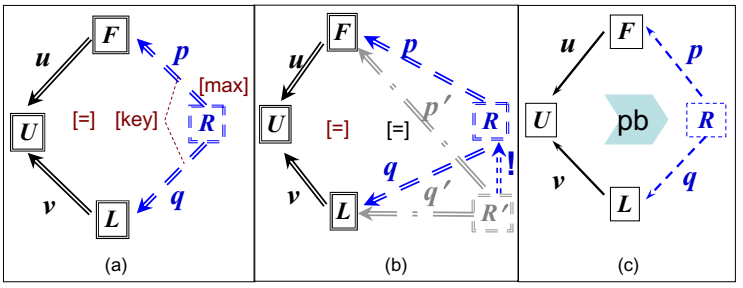


Fig. 28. Matching sets via arrows

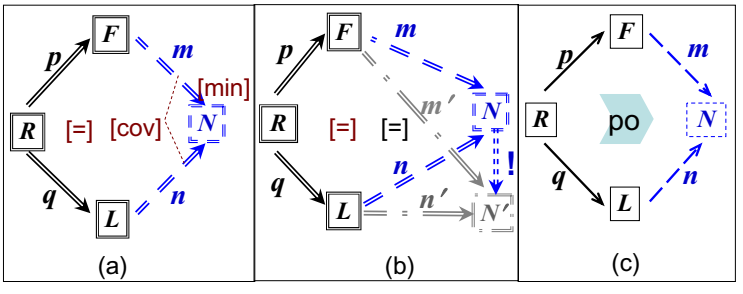


Fig. 29. Merging sets via arrows

$r_1 = r_2$ iff $r_1.p = r_2.p$ and $r_1.q = r_2.q$. That is, the pair of mappings (p, q) can identify the elements in set R, hence the name for the predicate. (In category theory, such families of mappings are called *jointly monic*). Note that any subset of set R defined in Fig. 27 will satisfy predicates [=] and [key]. Hence, to ensure that set R in Fig. 28 is indeed R defined in Fig. 27, we add one more predicate [max] stating R's maximality. Formally, it may be formulated as follows: for any other key span (p', q') as shown in Fig. 28(b), which makes the entire square commutative, there is a mapping $!: R' \to R$ such that $!;p = p'$ and $!;q = q'$.

Thus, we have reformulated the task of matching sets in terms of mappings, their composition and predicate [ke v y]. However, the latter also can be expressed via mappings and composition!

Suppose that span (p, q) is not required to be a key, but has the following property: for any other span (p', q') (also not assumed to be a key), which makes the entire square commutative, there is a *unique* mapping $!: R' \to R$ such that $!;p = p'$ and $!;q = q'$. This maximality property is distinct from that previously formulated by the uniqueness requirement, and this is what does the job. That is, we can prove that uniqueness of ! implies the [key] property of span (p, q). Given an element $r' \in R'$, let $f' = r'.p'$ and $l' = r'.q'$ be its "names". To ensure commutativity conditions: $!;p = p'$ and $!;q = q'$, function ! must map r' into any element r of R with the same names: $r.p = f'$ and $r.q = l'$. If span (p, q) is not a key, there may be several such elements r and hence several functions ! providing commutativity. Hence, ! is unique iff span (p, q) is a key.

Thus, we may replace predicates [key] and [max] of span (p, q) in Fig. 28(a) by the uniqueness property: for any other span (p', q') that makes the entire square commutative (Fig. 28b), there is a unique mapping $!: R' \to R$ such that $!; p = p'$ and $!; q = q'$. In category theory such properties are called *universal*. The entire matching construction can now be formulated in abstract terms as follows. Given a cospan (u, v), a span with the same feet is derived and together with the original cospan makes a commutative square with the universal property described above. An operation producing a universal span from a matching cospan is called *pullback* (because it pulls two arrows back). The result is shown in Fig. 28(c) which depicts abstract nodes and arrows (single lines) whose internal structure is invisible.

Is pullback indeed an operation, i.e., does it indeed result in a uniquely determined span? The answer is almost positive: the result of a pullback is defined up to isomorphism. The proof can be found in any CT-textbook, e.g., [69][Theorem 5.2.2], and essentially uses associativity of arrow composition. Other constructions based on universal properties are also defined up to isomorphism.

Merging. Our construction of merging sets can be processed in a similar way. Figure 29 presents the ideas in parallel to Fig. 28. Diagram predicate [cov] declared for cospan (m, n) says that the two mappings jointly cover the target, that is, any element $e \in N$ is either in the image of mapping m or n or both. We replace this predicate by the following universal property: for any other cospan (m', n') making the entire square commutative, there exists a unique mapping $!: N \to N'$ such that $m; ! = m'$ and $n; ! = n'$. Indeed, if set N would contain an element e beyond the union of images of m, n, mapping ! could map this e to any element of N' without destroying the commutativity conditions.

Thus, we can define set merge in terms of mappings, their composition and the universal property of minimality. The operation that takes a span and produces a cospan making a commutative square with the minimal universal property is called *pushout* (as it pushes arrows out). The construction is dual to the construction of pullback in the sense that all arrows in the diagrams are reversed, and universal maximality is replaced by universal minimality.[8] Particularly, the result of pushout is also defined up to isomorphism.

Summary. We have defined matching and merging sets via mappings (functions) between sets and their sequential composition. Of course, to define the latter, we still need the notion of element: composition of mappings $f: A \to B$ and $g: B \to C$ is defined by setting $x.(f; g) \stackrel{\text{def}}{=} (x.f).g$ for all elements $x \in A$. However, if we consider some universe of abstract objects (nodes) and abstract associatively composable mappings (arrows) between them, then we can define pullback and pushout operation as described above. Such graphs are called *categories* and, thus, the notions of match and merge can be defined for any category irrespective of the internal structure of its objects. The next sections provides precise definitions.

[8] It can be made perfectly dual if we formulate the predicate [cov] in a different way exactly dual to predicate [key].

B Graphs, Categories and Diagrams: A Primer

In this section we fix notation and terminology about graphs and categories. We also accurately define diagrams and diagram operations.

Graphs and graph mappings. A *(directed multi-)graph* $\boldsymbol{G}$ consists of a set of *nodes* $\boldsymbol{G}_0$ and a set of *arrows* $\boldsymbol{G}_1$ together with two functions $\partial_{\mathsf{x}}\colon \boldsymbol{G}_1 \to \boldsymbol{G}_0$, $\mathsf{x} = s, t$. For an arrow a we write $a\colon N \to N'$ if $\partial_{\mathsf{s}} a = N$ and $\partial_{\mathsf{t}} a = N'$. The set of all arrows $a\colon N \to N'$ is denoted by $\boldsymbol{G}(N, N')$ or, sometimes, by $(N \to N')$ if graph G is given by the context.

A *graph mapping (morphism)* $f\colon \boldsymbol{G} \to \boldsymbol{G}'$ is a pair of functions $f_i\colon \boldsymbol{G}_i \to \boldsymbol{G}'_i$, $i = 0, 1$, compatible with incidence of nodes and arrows: $\partial_{\mathsf{s}} f_1(a) = f_0(\partial_{\mathsf{s}} a)$ and $\partial_{\mathsf{t}} f_1(a) = f_0(\partial_{\mathsf{t}} a)$ for any arrow $a \in \boldsymbol{G}_1$.

A graph is *reflexive* if every node N has a special *identity* loop $\mathsf{1}_N\colon N \to N$. In other words, there is an operation $\mathsf{1}\_\colon \boldsymbol{G}_0 \to \boldsymbol{G}_1$ (with argument placed at the under-bar subscript) s.t. $\partial_{\mathsf{s}} \mathsf{1}_N = N = \partial_{\mathsf{t}} \mathsf{1}_N$ for any node N. If arrows are understood behaviorally (rather than structurally) as actions or transitions, identity loops may be also called *idle* (actions that do nothing and do not change the state). A reflexive graph *mapping (morphism)* is a graph mapping $f\colon \boldsymbol{G} \to \boldsymbol{G}'$ respecting identities: $f_1(\mathsf{1}_N) = \mathsf{1}_{f_0(N)}$ for any node $N \in \boldsymbol{G}_0$.

Categories and functors. A *category* $\boldsymbol{C}$ is a reflexive graph $|\boldsymbol{C}|$ with an operation of *arrow composition* denoted by ; (semi-colon): for any pair of sequentially composable arrows $a\colon M \to N$ and $b\colon N \to O$, a unique arrow $a; b\colon M \to O$ is defined. Composition is required to be *associative*: $(a; b); c = a; (b; c)$ for any triple a, b, c of composable arrows; and *unital*: $\mathsf{1}_{\partial_0(a)}; a = a = a; \mathsf{1}_{\partial_1(a)}$ for any arrow a.

Nodes in a category are usually called *objects*, and arrows are often called *morphisms*. Both a category $\boldsymbol{C}$ and its underlying graph $|\boldsymbol{C}|$ are normally denoted by the same letter $\boldsymbol{C}$. Thus, $\boldsymbol{C}_0$ and $\boldsymbol{C}_1$ denote the classes of all objects and all morphisms resp. The class of objects $\boldsymbol{C}_0$ can also be considered as a *discrete* category, whose only arrows are identities.

A category is called *thin* if for any pair of nodes (N, N') there is at most one arrow $a\colon N \to N'$. It is easy to see that a thin category is nothing but a preordered set with $N \leq N'$ iff there is an arrow $A\colon N \to N'$. Transitivity and reflexivity are provided by arrow composition and idle loops resp.

A *functor* $f\colon \boldsymbol{C} \to \boldsymbol{C}'$ between categories is a morphism of the underlying reflexive graphs that preserves arrow composition $f_1(a; b) = (f_1 a); (f_1 b)$.

Two-sorted graphs and 1.5-sorted categories. A *two-sorted graph* is a graph $\mathbb{G}$ whose arrows are classified into *horizontal* and *vertical*. That is, we have two disjoint graphs $\mathbb{G}_1^{\mathsf{h}}$ and $\mathbb{G}_1^{\mathsf{v}}$ sharing the same class of nodes $\mathbb{G}_0$. A two-sorted graph is *reflexive* if each node has both the *vertical* and the *horizontal* identity. A *two-sorted graph morphism (mapping)* is a graph mapping respecting arrow sorts.

A *two-sorted category* is a two-sorted reflexive graph $\mathbb{G}$ whose horizontal and vertical graphs are categories. Since horizontal composition (of matches) may be

problematic, in the paper we deal with *1.5-sorted categories*: two-sorted reflexive graphs in which only vertical arrows are composable and form a category.

Flat vs. deep graphs and categories. There are two ways of interpreting elements of graphs and categories that we will call *flat* and *deep*. According to a flat interpretation, elements of a graph do not have an internal structure, they are symbols/tokens that can be drawn on paper. A visual representation/picture of such a graph drawn on paper is practically equivalent to the graph itself (up to inessential visual nuances like sizes of nodes and thickness of arrows).

According to a deep interpretation, nodes of a graph are thought of as sets endowed with some structure, for example, plain sets with empty structure, or sets with a partial order (posets), or vector spaces, or flat graphs, or models over a given metamodel M. Correspondingly, arrows are thought of as structure-preserving mappings, e.g., functions between sets, monotone functions between posets, linear mappings between vector spaces, graph morphisms, symmetric deltas. As a rule, deep arrows are associatively composable and deep graphs are indeed categories, e.g., ***Sets*** (of sets and functions), ***Rels*** (of sets and relations), ***Posets*** (of posets and monotone functions), ***Graphs*** (of graphs and graph mappings), $\boldsymbol{Moddel}_{\mathsf{sym}}(M)$ (of M-models and symmetric deltas between them).

The description above is rough and overly simplistic. Making it more precise and intelligent needs a careful setting for logical and set-theoretical foundations, and goes far beyond our goals in the paper. Note, however, that we were talking about possible *interpretations* of elements constituting a category but the very definition of a category says nothing about "depth" of its objects and arrows.[9] Hence, any result proven for a general category (possessing some property P) is applicable to any flat or deep category (possessing P). For example, when we deal with category ***Modupd*** of models and updates, our results are applicable to any formalization of model and update as soon as we have a category.

As a rule, deep categories are infinite and cannot be drawn on paper (think of all sets or all M-models). However, we can draw a graph representing a small fragment of an infinite category, and further use and manipulate this representation in our reasoning about its deep referent. For example, nodes and arrows of a graph drawn on paper could refer to models and deltas, and operations over them correspond to synchronization procedures. Precise specification of these syntax-semantics relationships may be non-trivial. In the paper we deal with the following particular case of the issue: arity shapes of diagram operations are flat graphs whereas their carriers are deep. The next section provides an accurate formalization of this situation.

Diagrams. When different nodes or different arrows of a graph drawn on paper bear the same name, e.g., in Fig. 30(a1,a2), these names are *labels* "hung" on elements of the underlying graph rather than their unique names (the latter are unique identifiers of graph elements and cannot be repeated). Hence, in the

[9] It can be formalized in terms of so called *constructs* and *concrete* categories explained in book [70] (with care and elegance).

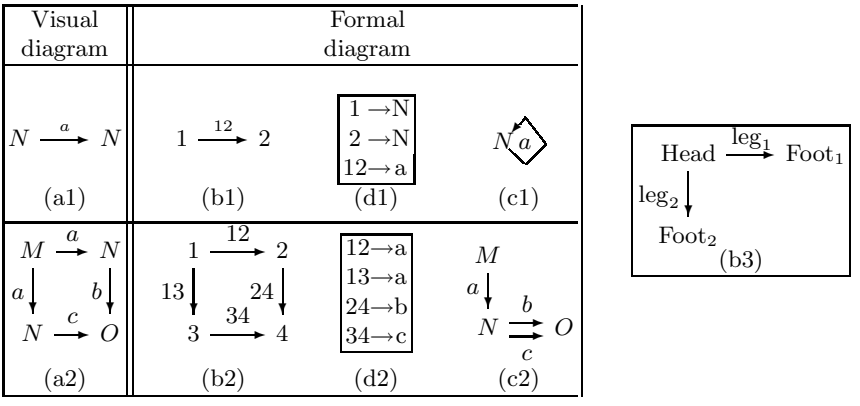


Fig. 30. Diagrams visually (a) and formally (b,d,c)

graphical image Fig. 30(a1), we have two unnamed different nodes (understood "flatly" as tokens) with the same label N. This label may be just another "flat" token, or the name (identifier) of a semantic object, e.g., a model; the formalization below does not take this into account (but in situations we deal with in the paper, labels are interpreted "deeply" as semantic objects).

It is convenient to collect all labels into a graph, and treat labeling as a graph mapping $D_1 : (b1) \to (c1)$ with (b1) and (c1) being graphs specified in Fig. 30(b1,c1) and mapping D_1 defined by table (d1), i.e., $D_1(1) = D_1(2) = N$, $D_1(12) = a$. Thus, image (a1) that we call a diagram consists of three components: graph (b1) called the *shape* of the diagram, graph (c1) called the *carrier*, and a graph mapping (d1) — the *labeling*. Since the shape and the carrier are actually referred to by the mapping, the latter alone can be called a *diagram* (it is a standard categorical terminology). Indeed, the graphical image — visual diagram shown in (a1) — is nothing but a compact presentation of mapping D_1 defined up to isomorphism of the shape.

For another example, visual diagram in Fig. 30(a2) encodes the formal diagram of shape (b2) in the carrier graph (c2) with labeling $D_2 : (b2) \to (c2)$ given by table (d2) (it is a graph morphism indeed).

What was earlier called a span in graph G, is actually a diagram $D : (b3) \to G$ with graph (b3) in Fig. 30 being the arity shape (the head of span D is node $D(\text{Head}) \in G$ etc.) Any span can be inverted: the *inverse* of D is another span $D^\dagger : (b3) \to G$ defined as follows: $D^\dagger(\text{leg}_1) = D(\text{leg}_2)$ and $D^\dagger(\text{leg}_2) = D(\text{leg}_1)$. Below we will call spans arity shapes (i.e. graphs isomorphic to (b3)) also *spans*.

Diagram operation over sorted graphs. Syntactically, a diagram operation is defined by its symbol (name), say, op, and a span of two-sorted graphs: $A_{\mathsf{op}} = (\mathbb{In}_{\mathsf{op}} \xleftarrow{p} \mathbb{IO}_{\mathsf{op}} \xrightarrow{q} \mathbb{Out}_{\mathsf{op}})$ whose legs are injections. The left foot specifies the *input* arity of the operation, the right one is the *output*, and the head is their intersection. For example, the operation of forward propagation considered above is specified by Fig. 31(a). The input arity is a span, the output arity is a cospan, and the head consists of two nodes.

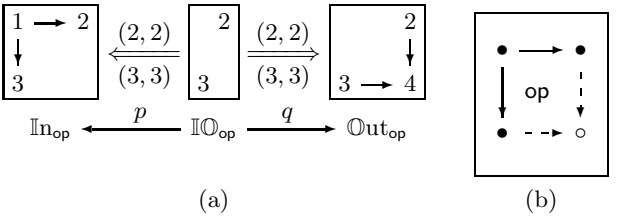


Fig. 31. Mechanism of diagram operations

We may merge both arities together (via pushout) and represent the arity span as a two-sorted graph $\mathbb{I}n\mathbb{O}ut$ with a designated subgraph $\mathbb{I}n$ of *basic* elements. For the forward propagation example, this construction is specified in Fig. 31(b): the basic subgraph is shown with black nodes and solid arrows, elements beyond the basic subgraph are white and dashed. We can restore graph $\mathbb{O}ut$ and the original arity span by subtracting graph $\mathbb{I}n$ from $\mathbb{I}n\mathbb{O}ut$ so that both formulations are equivalent. Previously we used the latter formulation because it is intuitive and compact.

Semantic interpretation of an operation is given by a pair $\sigma = (\mathbb{G}^\sigma, \mathsf{op}^\sigma)$ with $\mathbb{G}^\sigma$ a two-sorted graph being the *carrier* of the operation, and

$$\mathsf{op}^\sigma : (\mathbb{I}n_{\mathsf{op}} \to \mathbb{G}^\sigma) \to (\mathbb{O}ut_{\mathsf{op}} \to \mathbb{G}^\sigma),$$

the operation as such, being a total function between the functional spaces in round brackets. That is, any instantiation $i\colon \mathbb{I}n_{\mathsf{op}} \to \mathbb{G}^\delta$ of op's input in the carrier generates a unique instantiation $o\colon \mathbb{O}ut_{\mathsf{op}} \to \mathbb{G}^\delta$ of op's output, and we set $\mathsf{op}^\sigma(i) = o$. Moreover, both instantiations are required to be equal on their common part $\mathbb{I}\mathbb{O}_{\mathsf{op}}$, that is, $p; i = q; o$. In this way, the notion of diagram operation (its syntax and semantics) can be defined for any category (of "graphs").

The same idea is applicable to two-sorted graphs: both the shape and the carrier are two-sorted graphs and labeling must respect sorting. If we treat diagram Fig. 31(a2) as a two-sorted diagram, it would be incorrect because horizontal arrow 12 from the shape is mapped to vertical arrow a in the carrier.

Span composition. Categories are graphs, and hence the notion of a diagram, particularly, a span, applies to them as well. However, spans in categories are much more interesting than in graphs because we can sequentially compose them. Fig. 32 presents two consecutive spans between sets A, B, C. We may think of elements in the heads as bidirectional links and write $a \leftarrow r \rightarrow b$ for $r \in R_1$ if $p_1(r) = a$ and $q_1(r) = b$; and similarly for elements in R_2. If two such links $a \leftarrow r_1 \rightarrow b \in R_1$ and $(b \leftarrow r_2 \rightarrow c) \in R_2$ have a common end $b \in B$, we may compose them and form a new link $a \leftarrow r \rightarrow c$ denoted by $r_1; r_2$. By collecting together all such composed links, we form a new set R, which is equipped with two projections $(A \xleftarrow{p} R \xrightarrow{q} C)$. In addition, by the condition of compositionality, set R is equipped with another pair of projections $(R_1 \xleftarrow{p_2'} R \xrightarrow{q_1'} R_2)$ as shown in the figure, and it is easy to see that the upper square diagram is a pullback.

Note also that projections p and q are compositions $p = p'_2; p_1$ and $q = q'_1; q_2$. Now we may define the notion of span composition for any category having PBs, and achieve a remarkable generality.

There are however some hidden obstacles in this seemingly simple definition. Since pullbacks are defined up to iso(morphism), composition of spans is also defined up to iso. We may choose some canonical representatives in each of the iso classes, but then associativity of composition cannot be guaranteed. In fact, associativity would hold up to a canonic isomorphism too. It makes the universe of objects with arrows being spans a so called *bicategory* rather than a category, and essentially complicates the technical side.

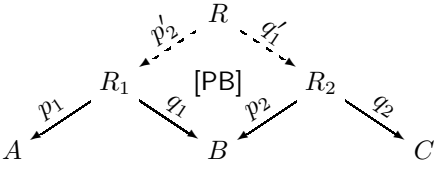


Fig. 32. Span composition

To avoid this, it is reasonable to consider spans up to isomorphism of their heads: it does not matter what are the OIDs of the head's elements. It is straightforward to check that composition of spans defined up to isomorphism of their heads is associative (details can be found in [71]).

Spans we deal with in the paper are special: their legs are injective mappings. It is known that if an input arrow in a PB-square is injective, the parallel output arrow is injective too ("monics are stable under PBs"). Hence, legs p'_2, q'_1 are injections, which implies that legs p, q are also injective as compositions of injections.

C Model Translation via Tiles

This section shows that model translation (MT) can be treated as a view computation, whose view definition is given by a corresponding metamodel mapping. Moreover, this construction can be modeled by tile operations, and gives rise to a well-known categorical construct called a *fibration*.

C.1 MT-Semantics and Metamodel Mappings

The MT-task is formulated as follows. Given two metamodels, $\boldsymbol{\mathcal{S}}$ (the source) and $\boldsymbol{\mathcal{T}}$ (the target), we need to design a procedure translating $\boldsymbol{\mathcal{S}}$-models into $\boldsymbol{\mathcal{T}}$-models. It can be formally specified as a function $\mathbf{f}: \mathbf{S} \to \mathbf{T}$ between the spaces of models (instances of the corresponding metamodels). The only role of metamodels in this specification is to define the source and the target spaces, and metamodels are indeed often identified with their model spaces [49,3,32]. However, a reasonable model translation $\mathbf{f}: \mathbf{S} \to \mathbf{T}$ should be compatible with model semantics. The latter is encoded in metamodels, and hence a meaningful translation should be somehow related to a corresponding relationship between the

metamodels. A simple case of such a relationship is when we have a mapping $f: \boldsymbol{\mathcal{T}} \to \boldsymbol{\mathcal{S}}$ between the metamodels. Indeed, if we want to translate $\boldsymbol{\mathcal{S}}$-model into $\boldsymbol{\mathcal{T}}$-models, the concepts specified in $\boldsymbol{\mathcal{T}}$ should be somewhere in $\boldsymbol{\mathcal{S}}$. The following example explains how it works.

Suppose that our source models consist of Person objects with attributes qName and phone: the former is complex and composed of a qualifier (Mr or Ms) and a string. The metamodel, $\boldsymbol{\mathcal{S}}$, is specified in the lower left quadrant of Fig. 33. Oval nodes refer to value types. The domain of the attribute 'qName' is a Cartesian product (note the label $\otimes$) with two projections 'name' and 'qual'. The target of the latter is a two-element enumeration modeled as the disjoint union of two singletons. Ignore dashed (blue with a color display) arrow and nodes for a while.

A simple instance of metamodel $\boldsymbol{\mathcal{S}}$ is specified in the upper left quadrant. It shows two Person-objects with names Mr.Lee and Ms.Lee (ignore blue elements again). Types (taken from the metamodel) are specified after colons and give rise to a mapping $t_A: A \to \boldsymbol{\mathcal{S}}$.

Another metamodel is specified in the lower right quadrant. Note labels [disj] and [cov] "hung" on the inheritance tree: they are diagram predicates (constraints) that require any semantic interpretation of node Actor (i.e., a set $[\![\mathit{Actor}]\!]$ of Actor-objects) to be exactly the disjoint union of sets $[\![\mathit{Male}]\!]$ and $[\![\mathit{Female}]\!]$.

We want to translate Person-models ($\boldsymbol{\mathcal{S}}$-instances) into Actor-models ($\boldsymbol{\mathcal{T}}$-instances). This intention makes sense if $\boldsymbol{\mathcal{T}}$-concepts are somehow "hidden" amongst $\boldsymbol{\mathcal{S}}$-concepts. For example, we may assume that Actor and Person refer to the same class in the real world.

The situation with Actor-concepts Male and Female is not so simple: they are not present in the Person-metamodel. However, although these concepts are not immediately specified in $\boldsymbol{\mathcal{S}}$, they can be *derived* from other $\boldsymbol{\mathcal{S}}$-concepts. We first derive new attributes /name and /qual by sequential arrow composition (see Fig. 33 with derived elements shown with dashed thin lines and with names prefixed by slash — a UML notation). Then, by the evident select-queries, we form two derived subclasses of class Person: mrPerson and msPerson.

Note that these two subclasses together with class Person satisfy the constraints [disj, cov] discussed above for metamodel $\boldsymbol{\mathcal{T}}$. It can be formally proved by first noting that enumeration {Mr,Mrs} is disjointly composed of singletons {Mr}, {Mrs}, and then using the property of Select queries (in fact, pullbacks) to preserve disjoint covering. That is, given (i) query specifications defining classes mrPesron, mrsPerson, and (ii) predicate declarations [disj, cov] for the triple ({Mr,Mrs},{Mr},{Mrs}), the same declarations for the triple (Person, mrPerson, mrsPerson) can be logically derived.

The process described above gives us an augmentation $Q[\boldsymbol{\mathcal{S}}] \supset \boldsymbol{\mathcal{S}}$ of the Person-metamodel $\boldsymbol{\mathcal{S}}$ with derived elements, where Q refers to the set of queries involved. Now we can relate Actor concepts Male and Female to derived Person-concepts mrPerson and mrsPerson. Formally, we set a total mapping $\boldsymbol{v}: \boldsymbol{\mathcal{T}} \to Q[\boldsymbol{\mathcal{S}}]$ that maps every $\boldsymbol{\mathcal{T}}$-element to a corresponding $Q[\boldsymbol{\mathcal{S}}]$-element. In Fig. 33, links constituting the mapping are shown by thin curly arrows. The mapping satisfies

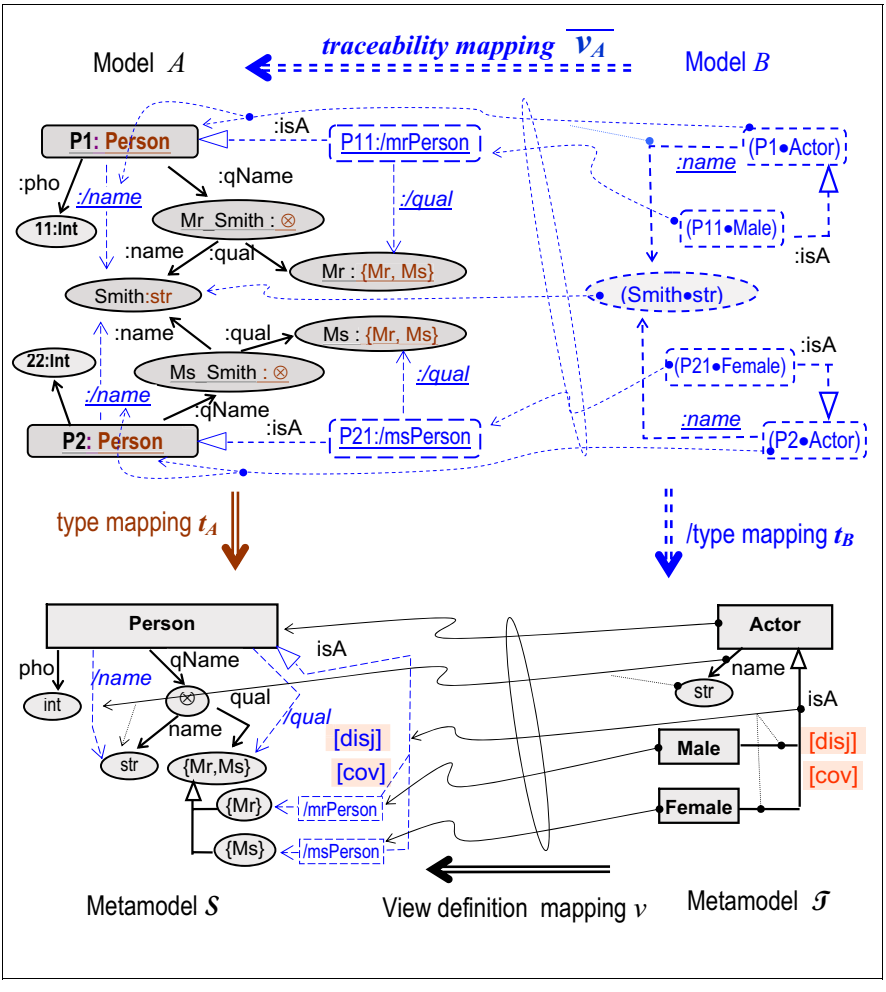


Fig. 33. Semantics of model translation via a metamodel mapping

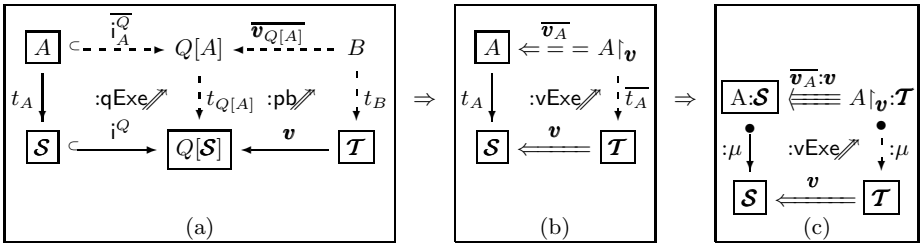


Fig. 34. Model translation via tile operations (the upper arrow in diagram (c) is derived and must be dashed but the Diagram software does not draw triple arrows)

two important requirements: (a) the structure of the metamodels (incidence of nodes and arrows, and the isA-hierarchy) is preserved; and (b) the constraints in metamodel $\boldsymbol{\mathcal{T}}$ are respected ([disj, cov]-configuration in $\boldsymbol{\mathcal{T}}$ is mapped to [disj,cov]-configuration in $\boldsymbol{\mathcal{S}}$).

Now we will show that data specified above are sufficient to automatically translate any $\boldsymbol{\mathcal{S}}$-model into a $\boldsymbol{\mathcal{T}}$-model via two tile operations.

C.2 MT via Tile Algebra

1) Query execution. Query specifications used in augmenting $\boldsymbol{\mathcal{S}}$ with derived elements can be executed for $\boldsymbol{\mathcal{S}}$-models. For example, each pair of some model's arrows typed with :qName and :name produces a composed arrow typed with :/name, and similarly any pair of some model's arrows :qName and :qual produces an arrow :/qual (these are not shown in the figure to avoid clutter). Then each object typed by :Person and having the value Mr along the arrow :/qual, is cloned and typed :/mrPerson.[10] The result is that the initial typing mapping $t_A\colon A \to \boldsymbol{\mathcal{S}}$ is extended to typing mapping $t_{Q[A]}\colon Q[A] \to Q[\boldsymbol{\mathcal{S}}]$, in which $Q[A]$ and $Q[\boldsymbol{\mathcal{S}}]$ denote augmentations of the model and the metamodel with derived elements.

This extended typing mapping is again structure preserving. Moreover, it is a *conservative extension* of mapping t_A in the sense that types of elements in A are not changed by $t_{Q[A]}$. Formally, the inverse image of submodel $\boldsymbol{\mathcal{S}} \subset Q[\boldsymbol{\mathcal{S}}]$ wrt. the mapping $t_{Q[A]}$ equals to A, and restriction of $t_{Q[A]}$ to A is again t_A.

The configuration we obtained is specified by the left square diagram in Fig. 34(a). Framed nodes and solid arrows denote the input for the operation of query execution, dashed arrows and non-framed nodes denote the result. Label [qExe] means that the entire square is produced by the operation; the names of arrows and nodes explicitly refer to query Q (whereas q is part of the label, not a separate name).

2) Retyping. The pair of mappings, typing $t_{Q[A]}\colon Q[A] \to Q[\boldsymbol{\mathcal{S}}]$, and view

[10] With a common semantics for inheritance, we should assign the new type label /mrPerson to the same object P1. To avoid multi-valued typing, inheritance is straightforwardly formalized by cloning the objects.

$Q[\boldsymbol{\mathcal{S}}] \xleftarrow{\boldsymbol{v}} \boldsymbol{\mathcal{T}}$, provide enough information for translating model $Q[A]$ into $\boldsymbol{\mathcal{T}}$-metamodel. All that we need to do is to assign to elements of $Q[A]$ new types according to the view mapping: if an element $e \in Q[A]$ has type $X = t_{Q[A]}(e) in Q[\boldsymbol{\mathcal{S}}]$ and $X = \boldsymbol{v}(Y)$ for some type $Y \in \boldsymbol{\mathcal{T}}$, we set the new type of e to be Y. For example, since $Q[A]$-element $P11$ in Fig. 33 has type mrPerson, which (according to the view mapping $\boldsymbol{v}$) corresponds to type Male in $\boldsymbol{\mathcal{T}}$, this elements must be translated into an instance of type Male; we denote it by $(P11 \bullet Male)$. If no such $\boldsymbol{\mathcal{T}}$-type Y exists, the element e is not translated and lost by the translation procedure (e.g., phones of Person-objects). Indeed, non-existence of Y means that the X-concept of metamodel $\boldsymbol{\mathcal{S}}$ is beyond the view defined by mapping $\boldsymbol{v}$ and hence all X-instances are to be excluded from $\boldsymbol{v}$-views.

Thus, translation is just retyping of some of $Q[A]$-elements by $\boldsymbol{\mathcal{T}}$-types, and hence elements of the translated model B are, in fact, pairs $(e, Y) \in Q[A] \times \boldsymbol{\mathcal{T}}$ such that $t_{Q[A]}(e) = \boldsymbol{v}(Y)$. In Fig. 33, such pairs are denoted by a bullet between the components, e.g., P1•Actor is a pair (P1,Actor) *etc.* If we now replace bullets by colons, we come to the usual notation for typing mappings. The result is that elements of the original model are retyped by the target metamodel according to the view mapping, and if B denotes the result of translation, we may write

$$B \cong \big\{(e, Y) \in Q[A] \times \boldsymbol{\mathcal{T}} \colon\ t_{Q[A]}(e) = \boldsymbol{v}(Y)\big\} \tag{1}$$

We use isomorphism rather than equality because elements of B should be objects and links rather than pairs of elements. Indeed, the translator should create a new OId for each pair appearing in the right part of (1).

First components of pairs specified in (1) give us a traceability mapping $\overline{\boldsymbol{v}_A} \colon B \to A$ as shown in Fig. 33. Second components provide typing mapping $t_B \colon B \to \boldsymbol{\mathcal{T}}$. The entire retyping procedure thus appears as a diagram operation specified by the right square in Fig. 34(a): the input of the operation is a pair of mappings $(t_{Q[A]}, \boldsymbol{v})$, and the output is another pair $(\overline{\boldsymbol{v}_{Q[A]}}, t_B)$. The square is labeled [pb] because equation (1) specifies nothing but an instance of pullback operation discussed in Sect. A.1.

Remark 7. If view $\boldsymbol{v}$ maps two different $\boldsymbol{\mathcal{T}}$-types $Y_1 \neq Y_2$ to the same $\boldsymbol{\mathcal{S}}$-type X, each element $e \in Q[A]$ of type X will gives us two pairs (e, Y_1) and (e, Y_2) satisfying the condition above and hence translation to $\boldsymbol{\mathcal{T}}$ would duplicate e. However, this duplication is reasonable rather than pathological: equality $\boldsymbol{v}(Y_1) = \boldsymbol{v}(Y_2) = X$ means that in the language of $\boldsymbol{\mathcal{T}}$ the type X simultaneously plays two roles (those described by types Y_1 and Y_2) and hence each X-instance in $Q[A]$ must be duplicated in the translation. Further examples of how specification (1) works can be found in [72]. They show that the pullback operation is surprisingly "smart" and provides an adequate and predictive model of retyping.[11]

[11] Since the construct of inverse image is also nothing but a special case of pullback, the postcondition for operation [qExe] stating that $t_{Q[A]}$ is a conservative extension can be formulated by saying that the square [qExe] is a pullback too. To be precise, if we apply pullback to the pair $(\mathsf{i}^Q_A, t_{Q[A]})$, we get the initial mapping t_A.

Constraints do matter. To ensure that view model B is a legal instance of the target metamodel $\boldsymbol{\mathcal{T}}$, view definition mapping $\boldsymbol{v}$ must be compatible with constraints declared in the metamodels. In our example in Fig. 33, the inheritance tree in the domain of $\boldsymbol{v}$ has two constraints [disj,cov] attached. Mapping $\boldsymbol{v}$ respects these constraints because it maps this tree into a tree (in metamodel $\boldsymbol{\mathcal{S}}$) that has the same constraints attached. Augmentation of model A with derived elements satisfies the constraints, $A \models [\mathsf{disj}] \wedge [\mathsf{cov}]$, because query execution (semantics) and constraint derivation machinery (pure logic, syntax) work in concert (the completeness theorem for the first order logic). Relabeling does nothing essential and model B satisfies the original constraint in $\boldsymbol{\mathcal{T}}$ as well (details can be found in [16]).

Arrow encapsulation. Query execution followed by retyping gives us the operation of view execution shown in Fig. 34(b). In the tile language, the outer tile [vExe] is the horizontal composition of tiles [qExe] and [pb]. Note that queries are "hidden" (encapsulated) within double arrows: their formal targets are ordinary models but in the detailed elementwise view their targets are models augmented with derived elements.

Diagram (c) present the operation in an even more encapsulated way. The top triple arrow denotes the entire diagram (b): the source and target nodes are models together with their typing mappings, and the arrow itself is the pair of mappings $(\boldsymbol{v}, \overline{\boldsymbol{v}_A})$. Although the source and the target of the triple arrow are typing mappings, we will follow a common practice and denote them by pairs (model:metamodel), e.g., $A{:}\boldsymbol{\mathcal{S}}$, leaving typing mappings implicit. Two vertical arrows are links, i.e., pairs $(A, \boldsymbol{\mathcal{S}})$, $(B, \boldsymbol{\mathcal{T}})$; a similar link from the top arrow to the bottom one is skipped. Diagram Fig. 34(c) actually presents a diagram operation: having a metamodel mapping $\boldsymbol{\mathcal{S}} \xLeftarrow{\boldsymbol{v}} \boldsymbol{\mathcal{T}}$ and a model $A{:}\boldsymbol{\mathcal{S}}$, view execution produces a model $A{\restriction_{\boldsymbol{v}}}{:}\boldsymbol{\mathcal{T}}$ along with a traceability mapping (triple arrow) $\overline{\boldsymbol{v}_A}{:}\boldsymbol{v}$ encoding the entire diagram Fig. 34(b). We will return to this construction later in Sect. D.3.

C.3 Properties of the View Execution Operation

The view execution operation has three remarkable properties.

1) Unitality. If a view definition is given by the identity mapping, view execution is identity as well, as shown by diagram Fig. 35(b1).

2) Compositionality. Suppose we have a pair of composable metamodel mappings $\boldsymbol{v}1 : \boldsymbol{\mathcal{T}} \Rightarrow \boldsymbol{\mathcal{S}}$ and $\boldsymbol{v}2 : \boldsymbol{\mathcal{U}} \to \boldsymbol{\mathcal{T}}$, which defines $\boldsymbol{\mathcal{U}}$ as a view of a view of $\boldsymbol{\mathcal{S}}$. Clearly, execution of a composed view is composed from the execution of components so that for any $\boldsymbol{\mathcal{S}}$-model A we should have

$$\overline{\boldsymbol{v}1;\boldsymbol{v}2_A} = \overline{\boldsymbol{v}2_B}\,;\, \overline{\boldsymbol{v}1_A} \text{ with } B \text{ standing for } A{\restriction_{\boldsymbol{v}1}}$$

as shown in Fig. 35(b2). Formal proof of this fact needs an accurate definition of query specifications (see [52] for details), and then it will be a standard exercise in categorical algebra (with so called Kleisli triples). Details will appear elsewhere.

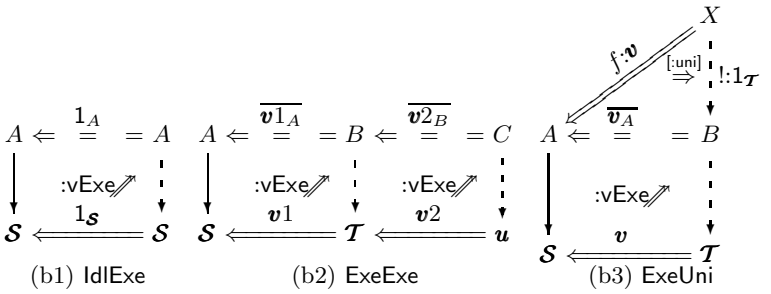


Fig. 35. Laws of the view execution mechanism

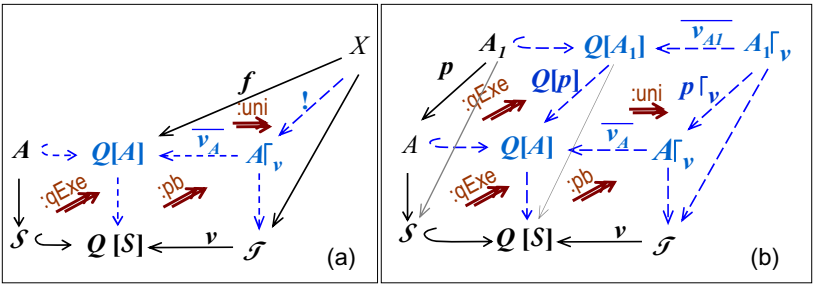


Fig. 36. Universal property of the view mechanism

3) Universality. Suppose we have a model X and a mapping $Q[A] \xleftarrow{f} X$ that maps some of X's elements to derived rather than basic elements of A as shown in Fig. 36(a). The mapping must be compatible with typing so that the outer right square is required to be commutative. Then owing to the universal properties of pullbacks, there is a uniquely defined mapping $A\restriction_{\boldsymbol{v}} \xleftarrow{!} X$ such that the triangles commute (note that mapping ! is a homogeneous model mapping over identity $1_{\boldsymbol{\mathcal{T}}} : \boldsymbol{\mathcal{T}} \to \boldsymbol{\mathcal{T}}$).

By encapsulating queries, i.e., hiding them inside double-arrows (see transition from diagram (a) to (b) in Fig. 34), we can formulate the property as shown in Fig. 35(b3), where arrows $f;\boldsymbol{v}$ and $!;1_{\boldsymbol{\mathcal{T}}}$ actually denote square diagrams whose vertical arrows are typing mappings and bottom arrows are pointed after semicolon.

View mechanism and updates. Universality of view execution has a remarkable consequence if queries are *monotonic*, i.e., preserve inclusion of datasets. Such queries have been studied in the database literature (e.g., [29]), and it is known that queries without negation are monotonic.

In our terms, a query Q is *monotonic* if any injective model mapping $A \xleftarrow{p} A_1$ between two $\boldsymbol{\mathcal{S}}$-models gives rise to an injective mapping $Q[A] \xleftarrow{Q[p]} Q[A_1]$ between models augmented with derived elements. This is illustrated by the left-upper square in Fig. 36(b). Applying retyping to models $A\restriction_{\boldsymbol{v}}$ and $A_1\restriction_{\boldsymbol{v}}$ provides

the rest of the diagram apart from arrow $p\restriction_{\boldsymbol{v}}$. To obtain the latter, we apply the universal property of $A\restriction_{\boldsymbol{v}}$ (specified in diagram (a)) to mapping $\overline{\boldsymbol{v}_{A_1}}; Q[p]$ (in the role of mapping f in diagram (a)), and get mapping $p\restriction_{\boldsymbol{v}}$. If the view definition mapping is injective, then traceability mappings are injective too (PBs preserve monics), and hence $p\restriction_{\boldsymbol{v}}$ is also injective. Thus, execution of views based on monotonic queries translates mappings as well. Moreover, if model updates are spans with injective legs, then view execution translates updates too: just add the other leg $q\colon A_1 \to A'$ and apply the same construction.

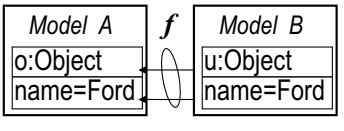
D Heterogeneous Model Mapping and Matching

Suppose that models A and B to be synchronized are instances of different metamodels, $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ respectively; we write $A{:}\boldsymbol{\mathcal{A}}$ and $B{:}\boldsymbol{\mathcal{B}}$. The metamodels may be essentially different, e.g., a class diagram and an activity diagram, which makes matching their instances structurally difficult. It even makes sense to reformulate the question ontologically: what is a match of non-similar models?

In Sect. 3.3 we modeled homogeneous matches by spans of homogeneous model mappings. We will apply the same idea to the heterogeneous case; hence, we first need to define heterogeneous model mappings.

D.1 Simple Heterogeneous Mappings

Model mappings are sets of links between model elements, and by *simple* mappings we mean those *not* involving derived elements. The first requirement for links to constitute a correct mapping is their compatibility with model structure: a class may be linked to a class, an attribute to an attribute etc. However, not all structurally correct mappings make sense.



Consider a mapping between two simple models in the inset figure. The mapping is structurally correct but it is not enough in the world of modeling, in which model elements have meaning encoded in metamodels. For example, Fig. 37(a) introduces possible metamodels for models A, B, and we at once see that sending an Employee to a Car is not meaningful. It could make sense if the concepts (classes) Employee and Car were "the same", in which case it must be explicitly specified by a corresponding mapping between the metamodels. Without such a mapping, the metamodels are not related and it may be incorrect to map an Employee to a Car. Thus, diagram Fig. 37(a) presents an incorrect model mapping.

An example of a correct model mapping is shown in diagram (b). We first build a metamodel mapping and map concept Employee to Person. It is then legitimate to map instances of Employee to instances of Person. Thus, a correct model mapping is a pair of mappings $(f, \mathbf{f})$ commuting with typing : $f; t_A = t_B; \mathbf{f}$.

Arrow encapsulation. An abstract schema of the example is shown in Fig. 37(c). As shown in Sect. 3.1, models are chains of graph mappings realizing

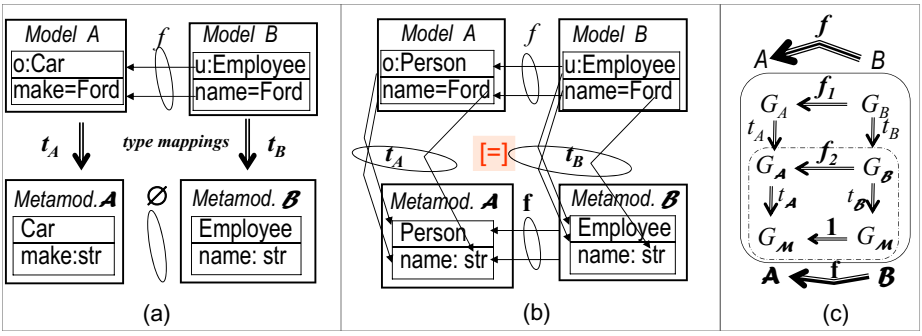


Fig. 37. Model mappings: incorrect (a), correct (b) and abstractly (c)

typing. Mapping $A \xleftarrow{f} B$ is a triple $(f_1, f_2, 1_{G_{\mathcal{M}}})$ commuting with typing mappings, i.e., f can be considered as a two-layer commutative diagram (framed by the bigger rounded rectangle). The lower layer (the smaller frame) is the metamodel mappings $\mathcal{B} \xleftarrow{\mathbf{f}} \mathcal{A}$. If metamodels $\mathcal{A}$ and $\mathcal{B}$ were instances of different meta-metamodels, $\mathcal{M}, \mathcal{N}$, we could still build a reasonable mapping f by introducing the third component $\mathcal{M} \xleftarrow{f_3} \mathcal{N}$ commuting with typing mappings of $\mathcal{M}$ and $\mathcal{N}$ to their common meta-metametamodel.

Irrespectively of the number of layers, a mapping between models $A \xleftarrow{f} B$ contains a corresponding mapping $\mathcal{B} \xleftarrow{\mathbf{f}} \mathcal{A}$ between metamodels (which contains a mapping between metametamodels). Hence, models and model mappings can be projected to metamodels and their mappings by erasing the upper layer. This projection is evidently a graph morphism μ: ***Modmap*** $\rightarrow$ ***MModmap*** from the graph of models and their simple mappings to the graph of metamodels and their simple mappings.

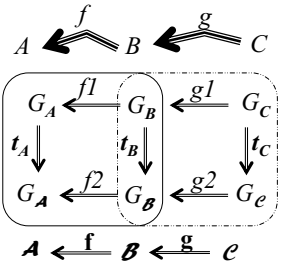


Fig. 38. Model mapping composition

Composition. Model mappings can be composed componentwise as shown in Fig. 38. The outer rectangle diagram is commutative as soon as the two inner squares are such. Hence, composition of two legal model mappings is again a legal model mapping. Associativity is evident, and the identity mapping consists of two identities $1_{G_A}: G_A \Rightarrow G_A$ and $1_{\mathcal{A}}: G_{\mathcal{A}} \Rightarrow G_{\mathcal{A}}$. Hence, graphs of (meta)models and their mappings introduced above are categories. Moreover, projection μ: ***Modmap*** $\rightarrow$ ***MModmap*** is evidently compatible with composition and identities and so is a functor.

D.2 Matching Heterogeneous Models

Consider a simple example shown in Fig. 39, where matching links between two models are set in a naive way, and compare it with naive match in Fig. 9(a) on p.111. The first peculiarity of match in Fig. 39 is that objects of different types

are matched (an Employee and a Personnel). Moreover, attributes are matched *approximately* meaning that their values somehow correspond but cannot be neither equal nor inequal because their relationship is more complex. Though intuitive, such matches do not conform to a type discipline, and their formal meaning is unclear.

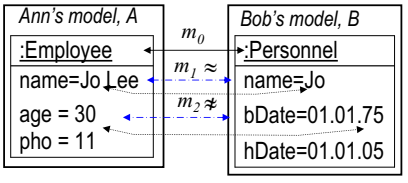


Fig. 39. Heterogeneous matching

Heterogeneous model mapping were defined above by including into them metamodel mappings. We may try to apply the same idea for heterogeneous matching: first match the metamodels, and then proceed with models. That is, we begin with making metamodels explicit and building a correspondence span between them as shown in Fig. 40(a).

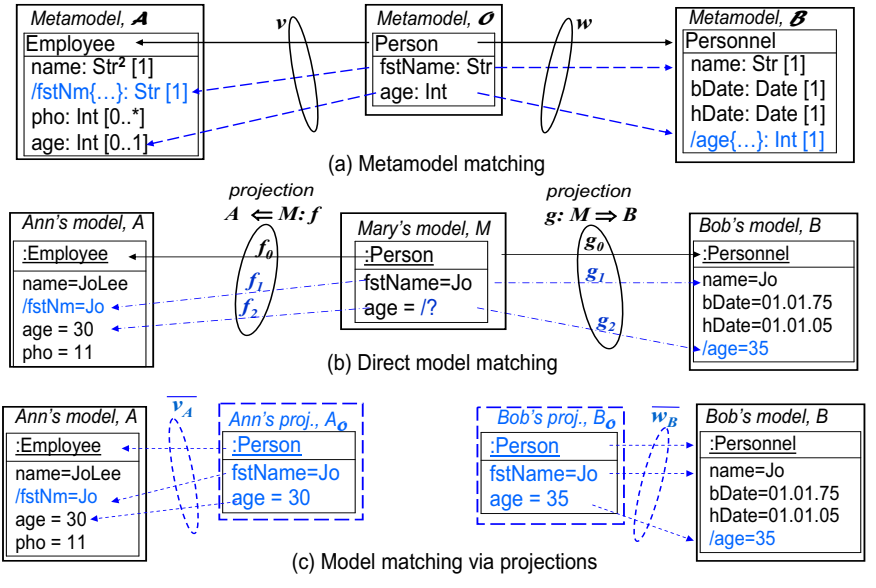


Fig. 40. Matching a heterogeneous pair of models

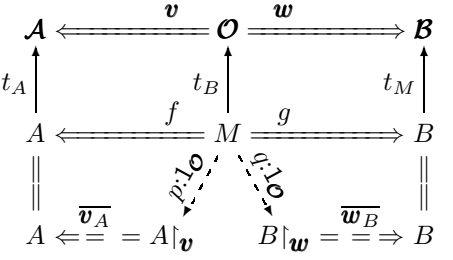
The head of the span, metamodel $\mathcal{O}$, specifies the concepts common for both metamodels (we will say a *metamodel overlap*), and the legs are projection mappings. A basic concept in one metamodel, e.g., attribute 'age' in metamodel $\mathcal{A}$, may be a derived concept in the other metamodel: there is no attribute 'age' in metamodel $\mathcal{B}$ but it can be derived from attribute 'bDate' with a corresponding query. Similarly, we may specify a query to the metamodel $\mathcal{A}$, which defines a new attribute 'fstNm' (firstName). (Ellipsis in figurative brackets near derived attributes in Fig. 40(a) refer to the corresponding query specifications.) Thus, the legs of a correspondence span may map elements in the head to derived elements in the feet.

Now we can reify the match in Fig. 39 by the span in Fig. 40(b). The feet are models A, B augmented with derived elements; the latter are computed by executing queries specified in the metamodels (recall that a derived element in a metamodel is a query definition). The legs are heterogeneous model mappings whose metamodel components are specified in diagram (a) (typing mappings are not shown). These mappings are similar to simple heterogeneous mappings considered in Sect. D.1 but may map to derived elements; we call them *complex*.

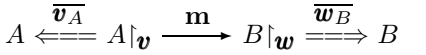
Metamodel mappings are view definitions that can be executed for models (Sect. C). By executing view $\boldsymbol{v}$ for model A, and view $\boldsymbol{w}$ for model B, we project the models to the space of $\boldsymbol{\mathcal{O}}$-instances as shown in Fig. 40(c): the view models are denoted by $A_{\boldsymbol{\mathcal{O}}} \stackrel{\text{def}}{=} A\restriction_{\boldsymbol{v}}$ and $B_{\boldsymbol{\mathcal{O}}} \stackrel{\text{def}}{=} B\restriction_{\boldsymbol{w}}$ (and their frames are dashed to remind us that these models are derived rather than set independently). We also call the views *projections* to the overlap space. Note that along with view models, view execution computes also traceability mappings $\overline{\boldsymbol{v}_A}$ and $\overline{\boldsymbol{w}_B}$.

There are evident mappings from the head M to projections $A_{\boldsymbol{\mathcal{O}}}$, $B_{\boldsymbol{\mathcal{O}}}$ (not shown in the figure to avoid clutter). The existence of these mappings can be formally proved by the universal property of pullbacks as described in Sect. C.3.

An abstract schema of the construction is shown in Fig. 41(a): the top row shows the metamodel overlap, the bottom row is the result of its execution, and the middle row is the correspondence span. Double-bodies of arrows remind us that mappings are complex, i.e., may map to derived elements in their targets.



(a) Extensional match



(b) General match

Fig. 41. From hetero- to homogeneous matches

Two slanted arrows are derived by the universal property of view traceability mappings (produced by pullbacks). Note that triple (M, p, q) is a homogeneous correspondence span in the space of $\boldsymbol{\mathcal{O}}$-models. It gives us an extensional match between models $A\restriction_{\boldsymbol{v}}$ and $B\restriction_{\boldsymbol{w}}$. We may add to this span non-extensional information (as discussed in Sect. 4.1) and come to diagram Fig. 41(b), in which arrow $\mathbf{m}$ denotes a general match between homogeneous models. Note that mappings $\overline{\boldsymbol{v}_A}$ and $\overline{\boldsymbol{w}_B}$ are derived whereas match $\mathbf{m}$ is an independent input datum.

D.3 Complex Heterogeneous Model Mappings

Simple heterogeneous model mappings defined above give rise to a functor $\mu\colon \boldsymbol{Modmap} \to \boldsymbol{MModmap}$. The goal of this section is to outline, semi-formally, how this description can be extended for complex mappings involving derived elements.

Let **QL** be a query language, that is, a signature of diagram operations over graphs. It defines a graph $\boldsymbol{MModmap}^{\mathbf{QL}}$ of metamodels and their complex mappings described in Sect. C. Similarly, we have graph $\boldsymbol{Modmap}^{\mathbf{QL}}$ of models and their complex mappings like, e.g., pairs mappings $(f, \boldsymbol{v})$ and $(g, \boldsymbol{w})$ shown in Fig. 40(b). (Recall that we actually deal with commutative square diagrams: $f; t_A = t_m; \boldsymbol{v}$ and $g; t_B = t_M; \boldsymbol{w}$.)

By encapsulating typing mappings inside nodes, and metamodel mappings inside arrows, we may rewrite the upper half of diagram Fig. 41(a) as shown in Fig. 42.

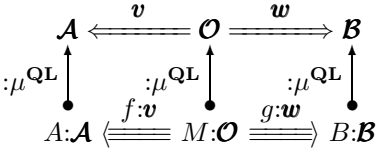


Fig. 42. Encapsulation of complex heterogeneous mappings

A warning about arrow notation is in order. Graph mappings in Fig. 37(c) are denoted by double arrows to distinguish them from links (single-line arrows), and diagrams of graph mappings are triple arrows. Complex mappings add one more dimension of encapsulation — derived elements, and hence mappings $\boldsymbol{v}$, $\boldsymbol{w}$ should be denoted by triple arrows while mappings-diagrams $f{:}\boldsymbol{v}$, $g{:}\boldsymbol{w}$ by quadruple arrows. To avoid this monstrous notation, we sacrifice consistency. It is partially restored by using bullet-end arrows for links: the latter may be thought of as arrows with "zero-line" bodies.

Thus, similarly to simple heterogeneous model mappings, complex ones contain complex metamodel mappings and hence there is a graph morphism

$$\mu^{\mathbf{QL}} : \boldsymbol{Modmap}^{\mathbf{QL}} \to \boldsymbol{MModmap}^{\mathbf{QL}}$$

(vertical links in Fig. 42 are its instances). We want to turn the two graphs above into categories (and $\mu^{\mathbf{QL}}$ into a functor), i.e., we need to define composition of complex mappings.

Composition of complex metamodel mappings is easy and amounts to term substitution. As mentioned above in Sect. C.2, with an accurate definition of a query langauge's syntax, compositionality of metamodel mappings is a routine exercise in categorical algebra (with the so called *Kleisli triples* [73]). It turns graph $\boldsymbol{MModmap}^{\mathbf{QL}}$ into a category (the *Kleisli category* of the *monad* defined by the query language).

Defining composition of complex model mappings is much harder because we need to compose query executions, i.e., application instances of operations rather than terms (definitions of operations). It can be done relatively easily for monotonic queries defined above on p.159 (details will appear elsewhere). Thus, if all queries are monotonic, graph $\boldsymbol{Modmap}^{\mathbf{QL}}$ can also be turned into a category, whose arrows are square diagrams similar to those shown in Fig. 38. We thus have a functor $\mu^{\mathbf{QL}} : \boldsymbol{Modmap}^{\mathbf{QL}} \to \boldsymbol{MModmap}^{\mathbf{QL}}$ that maps models and model mappings to their embedded metamodel parts.

The view mechanism is a "play-back" operation specified in Fig. 34(c) such that three laws in Fig. 35 are satisfied. Together these requirements mean that functor $\mu^{\mathbf{QL}}$ is a *(split) fibration* — a construct well-known in CT [74, Exercise 1.1.6]. The fibrational formulation of metamodeling (including the the view

mechanism) allows us to use many results of the rich theory of fibrations [74]. In a sense, it is a culmination of the concrete MMt branch of the paper: a multitude of complex data is encapsulated and cast in a very compact algebraic formulation.

Note that we did not formally prove the fibrational statement above. It is an observation suggested by our examples and semi-formal constructions in Sect. C and D rather than a theorem. To turn it into a theorem, we need a formal definition of queries and query execution, and then a formal specification of our considerations above; it is a work in progress. Part of this work is presented in [52] for the case of *functorial semantics* — a model is a functor *from* the metamodel to some semantic category of sets and mappings between them, which is dual to the usual metamodeling via a typing mapping (see beginning of Sect. 3.2).

An Introductory Tutorial on JastAdd Attribute Grammars

Görel Hedin

Lund University, Sweden
gorel@cs.lth.se

Abstract. JastAdd is an open-source system for generating compilers and other language-based tools. Its declarative specification language is based on reference attribute grammars and object-orientation. This allows tools to be implemented as composable extensible modules, as exemplified by JastAddJ, a complete extensible Java compiler. This tutorial gives an introduction to JastAdd and its core attribute grammar mechanisms, and how to use them when solving key problems in building language-based tools. A simple state machine language is used as a running example, showing the essence of name analysis, adding graphs to the abstract syntax tree, and computing circular properties like reachability. Exercises are included, and code for the examples is available online.

Keywords: attribute grammars, language-based tools, reference attributes, object-oriented model.

1 Introduction

JastAdd is a metacompilation system for generating language-based tools such as compilers, source code analyzers, and language-sensitive editing support. It is based on a combination of attribute grammars and object-orientation. The key feature of JastAdd is that it allows properties of abstract syntax tree nodes to be programmed declaratively. These properties, called *attributes*, can be simple values like integers, composite values like sets, and reference values which point to other nodes in the abstract syntax tree (AST). The support for reference-valued attributes is of fundamental importance to JastAdd, because they allow explicit definition of graph properties of a program. Examples include linking identifier uses to their declaration nodes, and representing call graphs and dataflow graphs. AST nodes are objects, and the resulting data structure, including attributes, is in effect an object-oriented graph model, rather than only a simple syntax tree.

While there are many technical papers on individual JastAdd mechanisms and advanced applications, this is the first tutorial paper. The goal is to give an introduction to JastAdd and its core attribute grammar mechanisms, to explain how to program and think declaratively using this approach, and to illustrate how key problems are solved.

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 166–200, 2011.

1.1 Object-Oriented Model

Figure 1 illustrates the difference from a traditional compiler where important data structures like symbol tables, flow graphs, etc., are typically separate from the AST. In JastAdd, these data structures are instead embedded in the AST, using attributes, resulting in an object-oriented model of the program. JastAdd is integrated with Java, and the resulting model is implemented using Java classes, and the attributes form a method API to those classes.

Attributes are programmed declaratively, using *attribute grammars*: Their values are stated using *equations* that may access other attributes. Because of this declarative programming, the user does not have to worry about in what order to evaluate the attributes. The user simply builds an AST, typically using a parser, and all attributes will then automatically have the correct values according to their equations, and can be accessed using the method API. The actual evaluation of the attributes is carried out automatically and implicitly by the JastAdd system.

The attribute grammars used in JastAdd go much beyond the classical attribute grammars defined by Knuth [Knu68]. In this tutorial, we particularly cover reference attributes [Hed00], parameterized attributes [Hed00, Ekm06], circular attributes [Far86, MH07] and collection attributes [Boy96, MEH09].

An important consequence of the declarative programming is that the object-oriented model in Fig. 1 becomes extensible. The JastAdd user can simply add new attributes, equations, and syntax rules. This makes it easy to extend languages and to build new tools as extensions of existing ones.

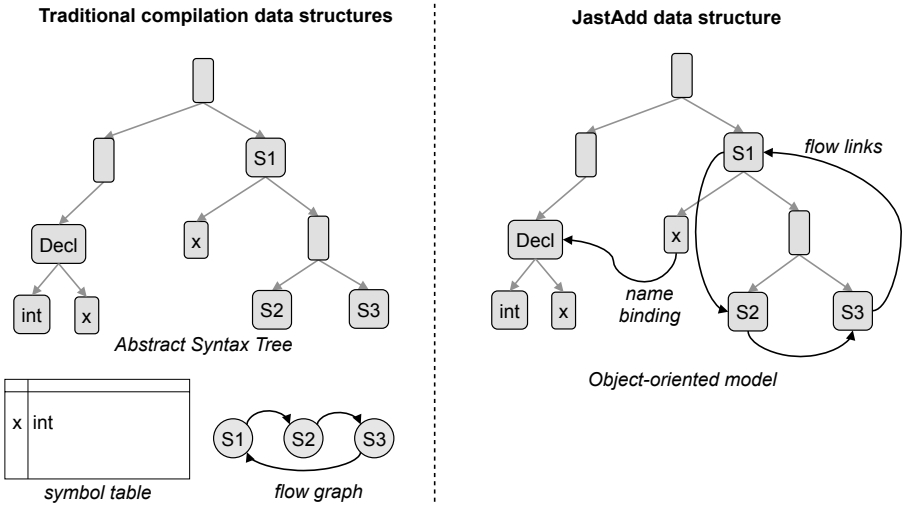


Fig. 1. In JastAdd, compilation data structures are embedded as reference attributes in the AST, resulting in an object-oriented model of the program

1.2 Extensible Languages and Tools

In JastAdd, the order of defining attributes and equations is irrelevant—their meaning is the same regardless of order. This allows the user to organize rules into modules arbitrarily, to form modules that are suitable for reuse and composition. Sometimes it is useful to organize modules based on compilation problems, like name analysis, type analysis, dataflow analysis, etc. Other times it can be useful to organize according to language constructs. As an example, in JastAddJ, an extensible Java compiler built using JastAdd [EH07b], both modularization principles are used, see Figure 2. Here, a basic compiler for Java 1.4 is modularized according to the classical compilation analyses: name analysis, type analysis, etc. In an extension to support Java 5, the modules instead reflect the new Java 5 constructs: the foreach loop, static imports, generics, etc. Each of those modules contain equations that handle the name- and type analyses for that particular construct. In yet further extensions, new computations are added, like non-null analysis [EH07a], separated into one module handling the Java 1.4 constructs, and another one handling the Java 5 constructs.

JastAdd has been used for implementing a variety of different languages, from small toy languages like the state machine language that will be used in this tutorial, to full-blown general-purpose programming languages like Java. Because of the modularization support, it is particularly attractive to use JastAdd to build extensible languages and tools.

1.3 Tutorial Outline

This tutorial gives an introduction to JastAdd and its core attribute grammar mechanisms. Section 2 presents a language for simple state machines that we will use as a running example. It is shown how to program towards the generated API for a language: constructing ASTs and using attributes. Basic attribu-

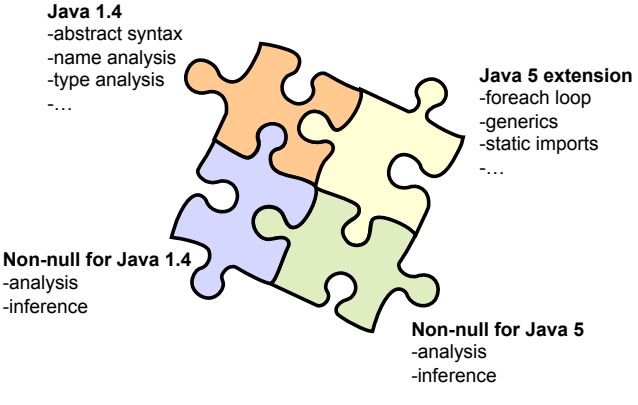


Fig. 2. Each component has modules containing abstract syntax rules, attributes, and equations. To construct a compiler supporting non-null analysis and inference for Java 5, all modules in the four components are used.

tion mechanisms are presented in Section 3, including synthesized and inherited attributes [Knu68], reference attributes [Hed00], and parameterized attributes [Hed00, Ekm06]. We show how name analysis can be implemented using these mechanisms. This section also briefly presents the underlying execution model.

The two following sections present more advanced mechanisms. Section 4 discusses how to define composed properties like sets using *collection attributes* [Boy96, MEH09]. These attributes are defined by the combination of values contributed by different AST nodes. We illustrate collection attributes by defining an explicit graph representation for the state machine, with explicit edges between state and transition objects. Section 5 discusses how recursive properties can be defined using *circular attributes* which are evaluated using fixed-point iteration [Far86, MH07]. This is illustrated by the computation of reachability sets for states. Finally, Section 6 concludes the tutorial.

The tutorial includes exercises, and solutions are provided in the appendix. We recommend that you try to solve the exercises on your own before looking at the solutions. The code for the state machine language and the related exercise solutions is available for download at `http://jastadd.org`. We recommend that you download it, and run the examples and solutions as you work through the tutorial. Test cases and the JastAdd tool are included in the download. See the README file for further instructions.

1.4 Brief Historical Notes

After Knuth's seminal paper on attribute grammars [Knu68], the area received an intense interest from the research community. A number of different evaluation algorithms were developed, for full Knuth-style AGs as well as for subclasses thereof. One of the most influential subclasses is Kastens' *ordered attribute grammars* (OAGs) [Kas80]. OAGs are powerful enough for the implementation of full programming languages, yet allow the generation of efficient static attribute evaluators. Influential systems based on OAGs include the *GAG* system which was used to generate front ends for Pascal and Ada [KHZ82, UDP$^+$82], and the *Synthesizer Generator* which supports incremental evaluation and the generation of interactive language-based editors [RT84]. For surveys covering this wealth of research, see Deransart *et al.* [DJL88], and Paakki [Paa95].

JastAdd belongs to a newer generation of attribute grammar systems based on reference attributes. Support for reference-like attributes were developed independently by a number of researchers: Hedin's reference attributes [Hed94, Hed00], Poetzsch-Heffter's occurrence attributes [PH97], and Boyland's remote attributes [Boy96].

Other landmark developments of strong importance for JastAdd include Jourdan's dynamic evaluation algorithm [Jou84], Farrow's circular attributes [Far86], Vogt, Swierstra and Kuiper's higher-order attributes [VSK89], and Boyland's collection attributes [Boy96].

In addition to JastAdd, there are several other current systems that support reference attributes, including Silver [WBGK10], Kiama [SKV09], and ASTER [KSV09]. While these systems use quite different syntax than JastAdd, and

support a partly different set of features, this tutorial can hopefully be of value also to users of these systems: the main ideas for how to think declaratively about reference attributes, and how to solve problems using them, still apply.

2 Running Example: A State Machine Language

As a running example, we will use a small state machine language. Figure 3 shows a sample state machine depicted graphically, and a possible textual representation of the same machine, listing all its states and transitions.

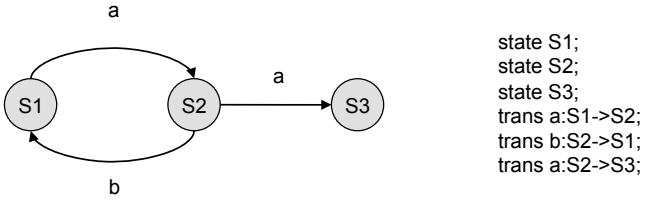


Fig. 3. A sample state machine and its textual representation

2.1 Abstract Grammar

Given the textual representation of a state machine, we would like to construct an object-oriented model of it that explicitly captures its graph properties. We can do this by first parsing the text into a syntax tree representation, and then add reference attributes to represent the graph properties. Fig. 4 shows a typical EBNF context-free grammar for the textual representation.

```
<statemachine> ::= <declaration>*;
<declaration> ::= <state> | <transition>;
<state> ::= "state" ID ";"
<transition> ::= "trans" ID ":" ID "->" ID ";";
ID = [a-zA-Z][a-zA-Z0-9]*
```

Fig. 4. EBNF context-free grammar for the state machine language

A corresponding abstract grammar, written in JastAdd syntax, is shown in Fig. 5. The nonterminals and productions are here written as *classes*, replacing alternative productions by subclassing: `StateMachine` is a class containing a list of `Declaration`s. `Declaration` is an abstract class, and `State` and `Transition` are its subclasses. The entities `Label`, etc. represent tokens of type `String`, and can be thought of as fields of the corresponding classes. An AST consists of a tree of objects of these classes. A parser that builds the AST from a text can be generated using an ordinary parser generator, building the AST in the semantic actions.

```
StateMachine ::= Declaration*;
abstract Declaration;
State : Declaration ::= <Label:String>;
Transition : Declaration ::=
  <Label:String> <SourceLabel:String> <TargetLabel:String>;
```

Fig. 5. JastAdd abstract grammar for the state machine language

2.2 Attributing the AST

To obtain an explicit object-oriented model of the graph, we would like to link each state object to the transition objects that has that state object as its source, and to link each transition object to its target state object. This can be done using reference attributes. Figure 6 shows the resulting object-oriented model for the example machine in Figure 3. We see here how the edges between state and transition objects are embedded in the AST, using reference attributes. Given this object-oriented model, we might be interested in computing, for example, reachability. The set of reachable states could be represented as an attribute in each `State` object. In sections 3, 4, and 5 we will see how these attributes can be defined.

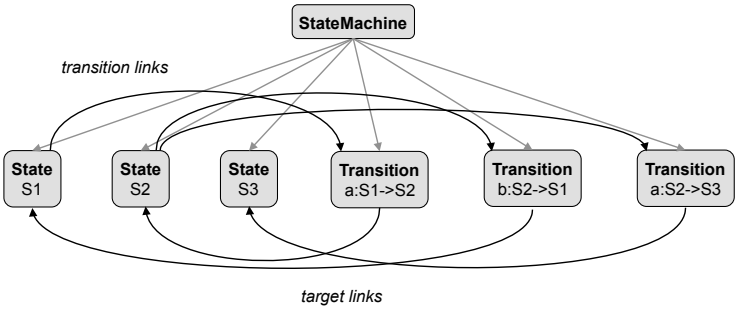


Fig. 6. The state machine graph is embedded in the object-oriented model

Exercise 1. In Figure 6, the objects are laid out visually to emphasize the AST structure. Make a new drawing that instead emphasizes the state machine graph. Draw only the `State` and `Transition` objects and the links between them, mimicking the layout in Figure 3.

2.3 Building and Using the AST

From the abstract grammar, JastAdd generates a Java API with constructors for building AST nodes and methods for traversing the AST. This API is furthermore augmented with methods for accessing the attributes. Figure 7 shows part of the generated API for the state machine language, including the attributes `target`, `transitions`, and `reachable` that will be defined in the coming sections.

```
class StateMachine {
  StateMachine();                          // AST construction
  void addDeclaration(Declaration node); // AST construction
  List<Declaration> getDeclarations();   // AST traversal
  Declaration getDeclaration(int i);     // AST traversal
}

abstract class Declaration {
}

class State extends Declaration {
  State(String theLabel);                  // AST construction
  String getLabel();                       // AST traversal
  Set<Transition> transitions();           // Attribute access
  Set<State> reachable();                  // Attribute access
}

class Transition extends Declaration {
  Transition(String theLabel, theSourceLabel, theTargetLabel);
                                           // AST construction
  String getLabel();                       // AST traversal
  String getSourceLabel();                 // AST traversal
  String getTargetLabel();                 // AST traversal
  State target();                          // Attribute access
}
```

Fig. 7. Parts of the API to the state machine model

Suppose we want to print out the reachable states for each state. For the small example in Figure 3, we would like to obtain the following output:

```
S1 can reach {S1, S2, S3}
S2 can reach {S1, S2, S3}
S3 can reach {}
```

meaning that all three states are reachable from S1 and S2, but no states are reachable from S3.

To program this we simply need to build the AST for the state machine, and then call the `reachable` attributes to print out the appropriate information. We do not need to do anything to attribute the AST—this is handled implicitly and automatically. To program the traversal of the AST in order to call the `reachable` attributes, it would be useful to add some ordinary Java methods to the AST classes. This can be done as a separate module using a JastAdd *aspect* as shown in Fig. 8.

The aspect uses *inter-type declarations* to add methods to existing classes. For example, the method `void StateMachine.printReachable() ...` means that

```
aspect PrintReachable {
  public void StateMachine.printReachable() {
    for (Declaration d : getDeclarations()) d.printReachable();
  }

  public void Declaration.printReachable() { }

  public void State.printReachable() {
    System.out.println(getLabel() + " can reach {" +
        listOfReachableStateLabels() + "}");
  }

  public String State.listOfReachableStateLabels() {
    boolean insideList = false;
    StringBuffer result = new StringBuffer();
    for (State s : reachable()) {
      if (insideList)
        result.append(", ");
      else
        insideList = true;
      result.append(s.getLabel());
    }
    return result.toString();
  }
}
```

Fig. 8. An aspect defining methods for printing the reachable information for each state

the method `void printReachable() ...` is added to the class `StateMachine`.[1]

We can now write the main program that constructs the AST and prints the reachable information, as shown in Fig. 9. For illustration, we have used the construction API directly here to manually construct the AST for a particular test program. For real use, a parser should be integrated. This is straightforward: build the AST in the semantic actions of the parsing grammar, using the same JastAdd construction API. Any Java-based parser generator can be used, provided it allows you to place arbitrary Java code in its semantic actions. In earlier projects we have used, for example, the LR-based parser generators *CUP* and *beaver*, and the LL-based parser generator *JavaCC*. For parser generators that automatically provide their own AST representation, a straightforward solution is to write a visitor that traverses the parser-generator-specific AST and builds the corresponding JastAdd AST.

[1] This syntax for inter-type declarations is borrowed from AspectJ [KHH+01]. Note, however, that JastAdd aspects support only static aspect-orientation in the form of these inter-type declarations. Dynamic aspect-orientation like pointcuts and advice are not supported.

```
public class MainProgram {
  public static void main(String[] args) {
    // Construct the AST
    StateMachine m = new StateMachine();
    m.addDeclaration(new State("S1"));
    m.addDeclaration(new State("S2"));
    m.addDeclaration(new State("S3"));
    m.addDeclaration(new Transition("a", "S1", "S2"));
    m.addDeclaration(new Transition("b", "S2", "S1"));
    m.addDeclaration(new Transition("a", "S2", "S3"));

    // Print reachable information for all states
    m.printReachable();
  }
}
```

Fig. 9. A main program that builds an AST and then accesses attributes

Exercise 2. Given the API in Fig. 7, write an aspect that traverses a state machine and prints out information about each state, stating if it is on a cycle or not. Hint: You can use the call `s.contains(o)` to find out if the set `s` contains a reference to the object `o`. What is your output for the state machine in Fig. 3? What does your main program look like?

3 Basic Attribution Mechanisms

We will now look at the two basic mechanisms for defining properties of AST nodes: *synthesized* and *inherited attributes*, which were introduced by Knuth in 1968 [Knu68]. Loosely speaking, synthesized attributes propagate information upwards in the AST, whereas inherited attributes propagate information downwards. The term *inherited* is used here for historical reasons, and its meaning is different from and unrelated to that within object-orientation.

3.1 Synthesized and Inherited Attributes

The value of an attribute a is defined by a directed equation $a = e(b_1, ..., b_n)$, where the left-hand side is an attribute and the right-hand side is an expression e over zero or more attributes b_k in the AST. In JastAdd, the attributes and equations are declared in AST classes, so we can think of each AST node as having a set of declared attributes, and a set of equations. Attributes are declared as either *synthesized* or *inherited*. A synthesized attribute is defined by an equation in the node itself, whereas an inherited attribute is defined by an equation in an ancestor node.

Most attributes we introduce will be synthesized. In the equation defining the attribute, we will use information in the node itself, say `E`, or by accessing its children, say, `F` and `G`. However, once in a while, we will find that the information

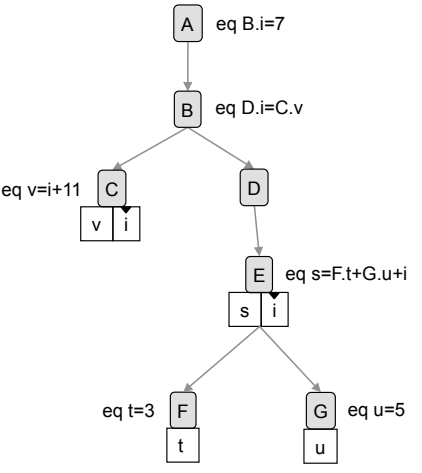


Fig. 10. The attributes `C.v`, `E.s`, `F.t`, and `G.u` are synthesized and have defining equations in the node they belong to. The attributes `C.i` and `E.i` are inherited (indicated by a downward-pointing black triangle), and are defined by equations in `A` and `B`, respectively. For `E.i`, the equation in `B` shadows the one in `A`, see the discussion below.

we need is located in the context of the `E` node, i.e., in its parent, or further up in the AST. In these cases, we will introduce an inherited attribute in `E`, capturing this information. It is then the responsibility of all nodes that could have an `E` child, to provide an equation for that inherited attribute.

In JastAdd, a shorthand is used so that the equation defining an inherited attribute, say `E.i`, does not have to be located in the immediate parent of `E`, but can be in any ancestor of `E`, on the way from the parent up to the root. If several of these nodes have an equation for `i`, the closest one to `E` will apply, shadowing equations further up in the AST. See Fig. 10. Thus, an equation *child.i* = *expression* actually defines the inherited *i* attribute for *all* nodes in the *child* subtree that declare the *i* attribute. This shorthand makes it possible to avoid cluttering the grammar with so called *copy rules*, i.e., equations that merely copy a value from a node to its children. Most attribute grammar systems have some kind of shorthand to avoid such copy rules. There are additional shorthands for this in JastAdd, for example allowing a single equation to be used to define an inherited attribute of *all* its children subtrees.

Exercise 3. What will be the values of the attributes in Fig. 10?

Exercise 4. An equation in node *n* for an inherited attribute `i` applies to the subtree of one of *n*'s children, say *c*. All the nodes in this subtree do not need to actually have an `i` attribute, so the equation applies only to those nodes that actually do. Which nodes in Fig. 10 are within the scope of an equation for `i`, but do not have an `i` attribute?

Exercise 5. In a correctly attributed AST, the attributes will have values so that all equations are fulfilled. How can the correct attribute values be computed? What different algorithms can you think of? (This is a difficult exercise, but worth thinking about.)

3.2 Reference Attributes

In JastAdd, synthesized and inherited attributes are generalized in several ways, as compared to the classical formulation by Knuth. The most important generalization is that an attribute is allowed to be a *reference* to an AST node. In this way, attributes can connect different AST nodes to each other, forming a graph. Furthermore it is allowed to use reference attributes inside equations, and to access the attributes of their referenced objects. This allows *non-local dependencies*: an attribute in one node can depend directly on attribute values in distant nodes in the AST. The dependencies do not have to follow the tree structure like in a classical AG. For example, if each use of an identifier has a reference attribute that points directly to the appropriate declaration node, information about the type can be propagated directly from the declaration to the use node.

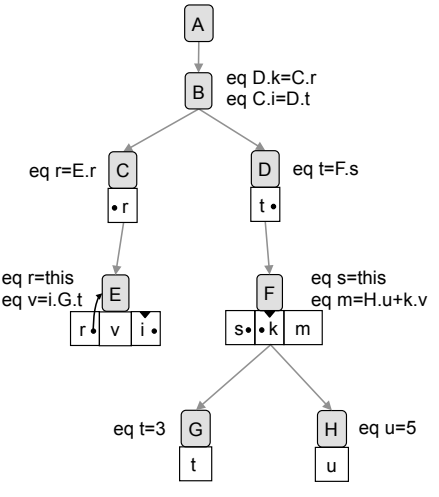


Fig. 11. Reference attributes are indicated with a small dot beside the name. An arrow from a dot to a node illustrates the value of the reference attribute: `E.r` refers to `E`, due to the equation `r=this` in `E`.

Reference attributes thus allow an AST to be extended to a graph in a declarative way. Also cyclic graphs can be defined, as in the example in Figure 11 (see also exercise 6). The example shows several possibilities for equations to access nodes and attributes, e.g.,

- `this`, meaning a reference to the node itself
- `k.v`, accessing the `v` attribute of the node referred to by `k`
- `i.G.t`, accessing the `t` attribute of the `G` child of the node referred to by `i`

Exercise 6. Draw the remaining reference attribute values in Figure 11. In what way is the graph cyclic? What are the values of the ordinary (non-reference) attributes? Give an example of non-local dependencies.

3.3 Parameterized Attributes

A second generalization in JastAdd is that attributes may have parameters. A parameterized attribute will have an unbounded number of values, one for each possible combination of parameter values. For example, we may define an attribute `lookup(String)` whose values are references to declarations, typically different for different parameter values. Conceptually, there is one value of `lookup` for each possible `String` value. In practice, only a few of these `lookup` values will actually be computed, because attribute evaluation is performed on demand (see Section 3.8).

By accessing a parameterized attribute via a reference attribute, complex computations can easily be delegated from one node to another. This is useful in, e.g., name analysis, where lookup can be delegated from a method to its enclosing class, and further on to superclasses, following the scope rules of the language.

3.4 Thinking Declaratively

When writing an attribute grammar, you should try to *think declaratively*, rather than to think about in which order things need to be computed. Think first what properties you would like the nodes to have to solve a particular problem. In the case of type checking, it would be useful if each expression node had a `type` attribute. The next step is to write equations defining these attributes. In doing so, you will need to solve subproblems that call for the addition of more attributes, and so on.

For example, to define the `type` attribute of an identifier expression, it would be useful to have an attribute `decl` that refers to the appropriate declaration node. You could then simply define the identifier's `type` as equal to the `type` of its declaration. The next problem is now to define the `decl` attribute. This problem would be easy to solve if all identifiers had a parameterized attribute `lookup(String)`, which returns a reference to the appropriate declaration node when supplied with the name of the identifier. The next problem is now in defining `lookup(String)`, and so on.

In adding a new attribute, you need to decide if it should be synthesized or inherited, i.e., if the node itself should define the attribute, or if the definition is delegated to an ancestor. If all the information needed is available inside the subtree rooted by the node, the attribute should be synthesized. If all the information is instead available outside this subtree, make the attribute inherited.

Finally, if both information inside and outside are needed, make the attribute synthesized, and introduce one or more inherited attributes to capture the information needed from outside.

As an example, consider the `type` attribute for expressions. Since the type will depend on what kind of expression it is, e.g., an identifier or an add node, the attribute should be synthesized. Similarly, the `decl` attribute should be synthesized since it depends on the identifier's name. The `lookup(String)` attribute, on the other hand, should be inherited since there is no information in the identifier node that is relevant for the definition of this attribute. The definition is in this case delegated to an ancestor node.

3.5 Integration with Java

The JastAdd specification language builds on top of Java. In using attributes, with or without parameters, we can view them as methods of AST nodes. Attributes are similar to abstract methods, and equations are similar to method implementations. In fact, when accessing attributes, we will use Java method call syntax, e.g., `a()`, and when we write an equation, the right-hand side is written either as a Java expression or as a Java method body.

Although ordinary Java code is used for the right-hand side of an equation, an important requirement is that it must not introduce any externally visible side effects such as changing fields of AST nodes or changing global data. I.e., its effect should be equivalent to the evaluation of a side-effect-free expression. The reason for this restriction is that equations represent definitions of values, and not effects of execution. As soon as an AST has been created, all its attributes automatically contain the correct values, according to their defining equations. The underlying attribute evaluator that accomplishes this will run the equation code, but the user does not have any explicit control over in what order the equations are run, or how many times they are run. For efficiency, the underlying machinery may memoize the values, i.e., run an equation just once, and store the value for subsequent accesses. And if a particular attribute is not accessed, its equation might not be run at all. Therefore, introducing externally visible side effects within the equations will not have a well-defined behavior, and may lead to very subtle bugs. The current JastAdd version (R20100416) does not check for side-effects in equations, but leaves this responsibility to the user. In principle, a number of static checks for this could be added, but this is an area of future work.

3.6 Example: Name Analysis for State Labels

In section 2 we discussed an attribute `target` for `Transition` objects, that should point to the appropriate target `State` object. This can be seen as a name analysis problem: We can view the states as declarations and the transitions as uses of those declarations. In addition to the `target` attribute we will define an analogous `source` attribute which points to the appropriate source `State` object. We start by declaring `target` and `source` as synthesized attributes of

`Transition`. This definition would be easy if we had a parameterized attribute `State lookup(String label)` that would somehow find the appropriate `State` object for a certain label. Since we don't have enough information in `Transition` to define `lookup`, we make it an inherited attribute. In fact, we will declare `lookup` as an attribute of the superclass `Declaration`, since it might be useful also to the `State` subclass, as we will see in exercise 8. By looking at the abstract grammar, we see that the `StateMachine` node can have children of type `Declaration`, so it is the responsibility of `StateMachine` to define `lookup`. (In this case, `StateMachine` will be the root of the AST, so there are no further ancestors to which the definition can be delegated.)

In `StateMachine`, we can define `lookup` simply by traversing the declarations, locating the appropriate state. To do this we will introduce a synthesized attribute `State localLookup(String label)` for `Declarations`. Fig. 12 shows the resulting grammar. We use a JastAdd aspect to introduce the attributes and equations using inter-type declarations.

```
aspect NameAnalysis {
  syn State Transition.source() = lookup(getSourceLabel()); // R1
  syn State Transition.target() = lookup(getTargetLabel()); // R2
  inh State Declaration.lookup(String label); // R3

  eq StateMachine.getDeclaration(int i).lookup(String label) { // R4
    for (Declaration d : getDeclarationList()) {
      State match = d.localLookup(label);
      if (match != null) return match;
    }
    return null;
  }

  syn State Declaration.localLookup(String label) = null; // R5

  eq State.localLookup(String label) = // R6
    (label.equals(getLabel())) ? this : null;
}
```

Fig. 12. An aspect binding each Transition to its source and target States

There are a few things to note about the notation used:

syn, inh, eq. The keywords `syn` and `inh` indicate declarations of synthesized and inherited attributes. The keyword `eq` indicates an equation defining the value of an attribute.

in-line equations. Rules R4 and R6 define equations using the `eq` keyword. But equations can also be given in-line as part of the declaration of a synthesized attribute. This is the case in rules R1, R2, and R5.

equation syntax. Equations may be written either using value syntax as in R1, R2, R5, and R6:

`attr = expr`,

or using method syntax as in R4:

`attr { ... return expr;}`

In both cases, full Java can be used to define the attribute value. However, as mentioned in Section 3.5, there must be no external side-effects resulting from the execution of that Java code. Even if R4 uses the method body syntax with a loop and an assignment, it is easy to see that there are no external side-effects: only the local variables `d` and `match` are modified.

equations for inherited attributes. R4 is an example of an equation defining an inherited attribute. The left-hand side of such an equation has the general form

```
A.getC().attr()
```

meaning that it is an equation in `A` which defines the `attr` attribute in the subtree rooted at the child `C` of the `A` node. If `C` is a list, the general form includes an argument `int i`:

```
A.getC(int i).attr()
```

meaning that the equation applies to the ith child of the list. The right-hand side of the equation is within the scope of `A`, allowing the API of `A` to be accessed directly. For example, in R4, the AST traversal method `getDeclarationList()` is accessed. The argument `i` is not used in this equation, since all the `Declaration` children should have the same value for `lookup`.

default and overriding equations. Default equations can be supplied in superclasses and overridden in subclasses. R5 is an example of a default equation, applying to all `Declaration` nodes, unless overridden in a subclass. R6 is an example of overriding this equation for the `State` subclass.

Exercise 7. Consider the following state machine:

```
state S1;
trans a: S1 -> S2;
state S2;
```

Draw a picture similar to Fig. 10, but for this state machine, i.e., indicating the location of all attributes and equations, according to the grammar in Fig. 12. Draw also the reference values of the `source` and `target` attributes. Check that these values agree with the equations.

Exercise 8. In a well-formed state machine AST, all `State` objects should have unique labels. Define a boolean attribute `alreadyDeclared` for `State` objects, which is true if there is a preceding `State` object of the same name.

Exercise 9. If there are two states with the same name, the first one will have `alreadyDeclared` = false, whereas the second one will have `alreadyDeclared` = true. Define another boolean attribute `multiplyDeclared` which will be true for both state objects, but false for uniquely named state objects.

3.7 More Advanced Name Analysis

The name analysis for the state machine language is extremely simple, since there is only one global name space for state labels. However, it illustrates the typical solution for name analysis in JastAdd: using inherited `lookup` attributes, and delegation to other attributes, like `localLookup`. This solution scales up to full programming languages. For example, to deal with block-structured scopes, the lookup attribute of a block can be defined to first look among the local declarations, and, if not found there, to delegate to the context, using the inherited lookup attribute of the block node itself. Similarly, object-oriented inheritance can be handled by delegating to a lookup attribute in the superclass. This general technique, using lookup attributes and delegation, is used in the implementation of the JastAddJ Java compiler. See [EH06] for details.

3.8 Attribute Evaluation and Caching

As mentioned earlier, the JastAdd user does not have to worry about in which order attributes are given values. The evaluation is carried out automatically. Given a well-defined attribute grammar, once the AST is built, all equations will hold, i.e., each attribute will have the value given by the right-hand side of its defining equation. From a performance or debugging perspective, it is, however, useful to know how the evaluation is carried out.

The evaluation algorithm is a very simple dynamic recursive algorithm, first suggested for Knuth-style AGs [Jou84], but which works also in the presence of reference attributes. The basic idea is that equation right-hand sides are implemented as recursive functions, and when an attribute is called, its defining equation is run. The function call stack takes care of evaluating the attributes in the right order.

The use of object-orientation, as in JastAdd, makes the implementation of the algorithm especially simple, representing both attributes and equations as methods: For synthesized attributes, ordinary object-oriented dispatch takes care of selecting the appropriate equation method. For inherited attributes, there is some additional administration for looking up the appropriate equation method in the parent, or further up in the AST.

Two additional issues are taken care of during evaluation. First, attribute values can be cached for efficiency. If the attribute is cached, its value is stored the first time it is accessed. Subsequent accesses will return the value directly, rather than calling the equation method. In JastAdd, attributes can be explicitly declared to be cached by adding the modifier `lazy` to their declaration. It is also possible to cache all attributes by using an option to the JastAdd system. Attributes that involve heavy computations and are accessed more than once (with the same arguments, if parameterized) are the best candidates for caching. For the example in Fig. 12 we could define `source` and `target` as cached if we expect them to be used more than once by an application:

```
...
syn lazy State Transition.source() = ...
syn lazy State Transition.target() = ...
...
```

The second issue is dealing with circularities. In a well-defined attribute grammar, ordinary attributes must not depend on themselves, directly or indirectly. If they do, the evaluation would end up in an endless recursion. Therefore, the evaluator keeps track of attributes under evaluation, and raises an exception at runtime if a circularity is found. Due to the use of reference attributes, there is no general algorithm for finding circularities by analyzing the attribute grammar statically [Boy05].

4 Composite Attributes

It is often useful to work with composite attribute values like sets, lists, maps, etc. In JastAdd, these composed values are often sets of node references. An example is the `transitions` attribute of `State`, discussed in Section 2. It is possible to define composite attributes using normal synthesized and inherited attributes. However, often it is simpler to use *collection* attributes. Collection attributes allow the definition of a composite attribute to be spread out in several different places in an AST, each contributing to the complete composite value. Collection attributes can be used also for scalar values like integers and booleans, see Exercise 13, but using them for composite values, especially sets, is more common.

4.1 Representing Composite Attributes by Immutable Objects

We will use objects to represent composite attribute values like sets. When *accessing* these attributes, great care must be taken to treat them as *immutable objects*, i.e., to only use their non-mutating operations. However, during the *construction* of the value, it is fine to use mutating operations. For example, an equation can construct a set value by successively adding elements to a freshly created set object. Figure 13 shows a simplified[2] part of the API of the Java class `HashSet`.

4.2 A Collection Attribute: `transitions`

A *collection attribute* [Boy96, MEH09] has a composite value that is defined as a combination of *contributions*. The contributions can be located anywhere in the AST. If we would use ordinary equations, we would need to define attributes that in effect traverse the AST to find the contributions. With collection attributes, the responsibility is turned around: each contributing node declares its contribution to the appropriate collection attribute.

[2] The actual API for `HashSet` has more general types for some parameters and returns booleans instead of void for some operations, and has many additional operations.

```
class HashSet<E> implements Set{
  public HashSet(); // Constructor, returns a new empty set.

  // Mutating operations
  public void add(E e); // Adds the element e to this object.
  public void addAll(Set<E> s); // Adds all elements in s to this object.

  // Non-mutating operations
  public boolean contains(T e); // Returns true if this set contains e.
  public boolean equals(Set<E> s); // Returns true if this set has the
                                   // same elements as s.
}
```

Fig. 13. Simplified API for the Java class HashSet

```
coll Set<Transition> State.transitions()       // R1
  [new HashSet<Transition>()] with add;

Transition contributes this                    // R2
  when source() != null
  to State.transitions()
  for source();
```

Fig. 14. Defining `transitions` as a collection attribute

Fig. 14 shows how to define the `transitions` attribute as a collection.

Rule R1 declares that `State` objects have a collection attribute `transitions` of type `Set<Transition>`. Its initial value (enclosed by square brackets) is `new HashSet<Transition>()`, and contributions will be added with the method `add`.

Rule R2 declares that `Transition` objects contribute themselves (`this`) to the `transitions` collection attribute of the `State` object `source()`, but only when `source()` is not equal to `null`.

We can note that the definition of `transitions` involves only the two node classes `State` and `Transition`. If we had instead used ordinary synthesized and inherited attributes to define `transitions`, we would have had to propagate information through `StateMachine`, using additional attributes. The collection attribute solution thus leads to a simpler solution, as well as less coupling between syntax node classes.

Exercise 10. Define an attribute `altTransitions` that is equivalent to `transitions`, but that uses ordinary synthesized and inherited attributes instead of collection attributes. Compare the definitions.

Via the `transitions` attribute, we can easily find the successor states of a given state. To obtain direct access to this information, we define an attribute `successors`. Figure 15 shows the definition of `successors` as an ordinary

```
syn Set<State> State.successors() {
  Set<State> result = new HashSet<State>();
  for (Transition t : transitions()) {
    if (t.target() != null) result.add(t.target());
  }
  return result;
}
```

Fig. 15. Defining `successors`, by using `transitions`.

synthesized attribute, making use of `transitions`. An alternative definition would have been to define `successors` independently of `transitions`, using a collection attribute.

Exercise 11. Define an attribute `altSuccessors` that is equivalent to `successors`, but that uses a collection attribute. Compare the definitions.

4.3 Collection Attribute Syntax

Figure 16 shows the syntax used for declaring collection attributes and contributions. For the *collection-attribute-declaration*, the *initial-object* should be a Java expression that creates a fresh object of type *type*. The *contributing-method* should be a one-argument method that mutates the *initial-object*. It must be *commutative*, i.e., the order of calls should be irrelevant and result in the same final value of the collection attribute. Optionally, a *rootclass* can be supplied, limiting the contributions to occur in the AST subtree rooted at the closest *rootclass* object above or at the *nodeclass* object in the AST. If no *rootclass* is supplied, contributions can be located anywhere in the AST.

In the *contribution-declaration*, the *expr* should be a Java expression that has the type of the argument of the *contributing-method*, as declared in the corresponding collection declaration (the one for *collection-nodeclass.attr()*). In the example, there is an `add` method in `Set<Transition>` which has the argument type `Transition`, so this condition is fulfilled. There can be one or more such contributions, separated by commas, and optionally they may be conditional, as specified in a `when` clause. The expression *ref-expr* should be a reference to a *collection-nodeclass* object. Optionally, the contribution can be added to a whole set of collection attributes by using the `each` keyword, in which case *ref-expr* should be a set of *collection-nodeclass* objects, or more precisely, it should be an object implementing Java's interface `Iterable`, and contain objects of type *collection-nodeclass*.

Exercise 12. Given the `successors` attribute, define a `predecessors` attribute for `State`, using a collection attribute. Hint: use the `for each` construct in the contribution.

Exercise 13. Collection attributes can be used not only for sets, but also for other composite types, like maps and bags, and also for scalar types like integers.

collection-attribute-declaration ::=
  'coll' *type nodeclass* '.' *attr* '()'
  '[' *initial-object* ']'
  'with' *contributing-method*
  ['root' *rootclass*]

contribution-declaration ::=
  contributing-nodeclass 'contributes'
  (*expr* ['when' *cond*] , ',')$^+$
  'to' *collection-nodeclass* '.' *attr* '()'
  'for' ['each'] *ref-expr*

Fig. 16. Syntax for collection attributes and contributions

Primitive types, like `int` and `boolean` in Java, need, however, to be wrapped in objects. Define a collection attribute `numberOfTransitions` that computes the number of transitions in a state machine.

Exercise 14. Define a collection attribute `errors` for `StateMachine`, to which different nodes in the AST can contribute strings describing static-semantic errors. Transitions referring to missing source and target states are obvious errors. What other kinds of errors are there? Write a collection declaration and suitable contributions to define the value of `errors`.

For more examples of collection attributes, see the *metrics* example, available at `jastadd.org`. This example implements Chidamber and Kemerer's metrics for object-oriented programs [CK94]. The implementation is done as an extension to the JastAddJ Java compiler, and makes heavy use of collection attributes for computing the different metrics. Collection attributes are also used in the *flow analysis* example at `jastadd.org`, as described in [NNEHM09]. Here, *predecessors* in control-flow graphs, and *def* and *use* sets in dataflow, are defined using collection attributes.

4.4 Evaluation of Collection Attributes

When accessing a collection attribute, JastAdd automatically computes its value, based on the existing contribution declarations. In general, this involves a complete traversal of the AST to find the contributions, unless the scope of the collection is restricted, using a `root` clause in the collection declaration. To improve performance, several collection attributes can be computed in the same traversal, either completely or partially. Given that a particular instance c_i of a collection attribute c is accessed, the default behavior of JastAdd is to partially compute all instances of c, so that further traversal of the AST is unnecessary when additional instances of c are accessed. The algorithm used is called *two-phase joint evaluation* [MEH09]. It is sometimes possible to achieve further performance improvements by using other algorithm variants. For example, the evaluation of several different collection attributes can be grouped, provided that they do not depend on each other. See [MEH09] for more details.

5 Circular Attributes

Sometimes, the definition of a property is *circular*, i.e., depending ultimately on itself: When we write down a defining equation for the property, we find that we need the same property to appear at the right-hand side of the equation, or in equations for attributes used by the first equation. In this case, the equations cannot be solved by simple substitution, as for normal synthesized and inherited attributes, but a fixed-point iteration is needed. The variables of the equations are then initialized to some value, and assigned new values in an iterative process until a solution to the equation system is found, i.e., a *fixed point*.

The `reachable` attribute of `State` is an example of such a circularly defined property. In this section we will first look at how this property can be formulated and solved mathematically, and then how it can be programmed using JastAdd.

5.1 Circularly Defined Properties

To define reachability for states mathematically, suppose first that the state machine contains n states, $s_1..s_n$. Let $succ_k$ denote the set of states that can be reached from s_k through one transition. The set of reachable states for s_k, i.e., the set of states that can be reached via any number of transitions from s_k, can then be expressed as follows:

$$reachable_k = succ_k \cup \bigcup_{s_j \in succ_k} reachable_j$$

We will have one such equation for each state $s_k, 1 \leq k \leq n$. If there is a cycle in the state machine, the equation system will be cyclic, i.e., there will be some *reachable* set that (transitively) depends on itself. We can compute a solution to the equation system using a *least fixed-point iteration*. I.e., we use one *reachable* variable for each state, to which we initially assign the empty set. Then we interpret the equations as assignments, and iterate these assignments until no *reachable* variable changes value. We have then found a solution to the equation system. The iteration is guaranteed to terminate if we can place all possible values in a lattice of finite height, and if all the assignments are monotonic, i.e., if they never decrease the value of any *reachable* variable.

In this case, the values are sets of states, and they can be arranged in a lattice with the empty set at the bottom and the set of all states in the state machine at the top. Fig. 17 shows the lattice for the state machine of Fig. 3. The lattice will be of finite height since the number of states in any given state machine is finite. The assignments will be monotonic since the union operator can only lead to increasing values in the lattice. Because we start at the bottom (the empty set), we are furthermore guaranteed to find the *least* fixed point, i.e., the variables will stay at the lowest possible points in the lattice. If we have a cycle in the state machine, there may be additional uninteresting fixed points, for example by assigning the full set of states to *reachable* for all states on the cycle.

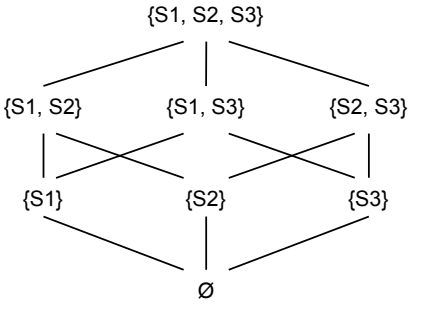


Fig. 17. The sets of states for the state machine of Fig. 3 are arranged in a lattice

Exercise 15. For the state machine of Fig. 3, write down all the equations for *reachable*. Which are the variables of the equation system?

Exercise 16. What is the (least) solution to this equation system? Are there any more (uninteresting) solutions?

Exercise 17. Construct a state machine for which there is more than one solution to the equation system. What would be the least solution? What would be another (uninteresting) solution?

5.2 Circular Attributes

In JastAdd, we can program circular properties like *reachable* by explicitly declaring the attribute as `circular`. and stating what initial value to use. The attribute will then automatically be evaluated using fixed-point iteration. Fig. 18 shows the definition of the attribute `reachable` for states.

```
syn Set<State> State.reachable() circular [new HashSet<State>()]; // R1

eq State.reachable() { // R2
  HashSet<State> result = new HashSet<State>();
  for (State s : successors()) {
    result.add(s);
    result.addAll(s.reachable());
  }
  return result;
}
```

Fig. 18. Defining `reachable` as a circular attribute

Things to note:

syntax. Synthesized, inherited and collection attributes can be declared as circular by adding the keyword `circular` after the attribute name. For synthesized and inherited attributes, an initial value also needs to be supplied,

surrounded by square brackets as shown in the example above. For collection attributes, the initial object is used as the initial value.

caching. Circular attributes are automatically cached, so adding the keyword `lazy` has no effect.

equals method. The types used for circular attributes must have a Java `equals` method that tests for equality between two attribute values.

value semantics. As usual, it is necessary to treat any accessed attributes as values, and to not change their contents. In the example, we set `result` to a freshly created object, and it is therefore fine to mutate `result` inside the equation. Note that if we instead had initialized `result` to the set of successors, we would have had to be careful to set `result` to a fresh clone of the `successors` object.[3]

termination. For attributes declared as circular, it would be nice to have static checks for the requirements of equation monotonicity and finite-height lattices. Currently, JastAdd does not support any such checks, but leaves this as a responsibility of the user. This means that if these requirements are not met, it may result in erroneous evaluation or non-termination during attribute evaluation. Boyland's APS system provides some support in this direction by requiring circular attributes to have predefined lattice types, like *union* and *intersection* lattices for sets, and *and* and *or* lattices for booleans [Boy96]. Similar support for JastAdd is part of future work.

Attributes that are not declared as circular, but which nevertheless happen to be defined circularly, are considered erroneous. To statically check for the existence of such attributes is, in general, an undecidable problem in the presence of reference attributes [Boy05]. In JastAdd, such circularities are detected dynamically, raising an exception at evaluation time.

Exercise 18. Define an attribute `altReachable` that is equivalent to `reachable`, but that uses a circular collection attribute. Hint: make use of the `predecessors` attribute defined in exercise 12.

For more examples of JastAdd's circular attributes, you may look at the *flow analysis* example at `jastadd.org` where intraprocedural control flow and dataflow is defined as an extension to the JastAddJ Java compiler, as described in [NNEHM09]. Here, the *in* set is defined as a circular attribute, and the *out* set as a circular collection attribute. In [MH07], there are examples of defining the properties *nullable*, $first$, and $follow$ for nonterminals in context-free grammars, using JastAdd circular attributes. The *nullable* property is defined using a boolean circular attribute, and the two others as set-valued circular attributes. A variant of $follow$ is defined in [MEH09] using circular collection attributes.

[3] In order to avoid having to explicitly create fresh objects each time a new set value is computed, we could define an alternative Java class for sets with a larger nonmutating API, e.g., including a `union` function that automatically returns a new object if necessary. Such an implementation could make use of persistent data structures [DSST86], to efficiently represent different values.

6 Conclusions and Outlook

In this tutorial we have covered central attribution mechanisms in JastAdd, including synthesized, inherited, reference, parameterized, collection, and circular attributes. With these mechanisms you can address many advanced problems in compilers and other language tools. There are some additional mechanisms in JastAdd that are planned to be covered in a sequel of this tutorial:

Rewrites [EH04], allow sub ASTs to be replaced conditionally, depending on attribute values. This is useful when the AST constructed by the parser is not specific enough, or in order to normalize language constructs to make further compilation easier.

Nonterminal attributes [VSK89] allow the AST to be extended dynamically, defining new AST nodes using equations. This is useful for macro expansion and transformation problems. In the JastAddJ Java compiler, nonterminal attributes are used for adding nodes representing instances of generic types [EH07b].

Inter-AST references [ÅEH10] allow nodes in a new AST to be connected to nodes in an existing AST. This is useful when creating transformed ASTs: nodes in the transformed AST can have references back to suitable locations in the source AST, giving access to information there.

Interfaces. Attributes and equations can be defined in interfaces, rather than in AST classes, allowing reuse of language independent computations, and supporting connection to language independent tools.

Refine. Equations in existing aspects can be *refined* in new aspects. This is similar to object-oriented overriding, but without having to declare new subclasses. Refines are useful for adjusting the behavior of an aspect when reusing it for a new language or tool.

The declarative construction of an object-oriented model is central when programming in JastAdd. The basic structure is always the abstract syntax tree (AST), but through the reference attributes, graphs can be superimposed. In this tutorial we have seen this through the addition of the `source` and `target` edges, and the `transitions`, `successors`, and `reachable` sets. Similar techniques are used to implement compilers for programming languages like Java. Here, each use of an identifier can be linked to its declaration, each class declaration to its superclass declaration, and edges can be added to build control-flow and dataflow graphs. Once these graphs have been defined, further attribute definitions are often made in terms of those graph structures rather than in terms of the tree structure of the AST. An example was defining `transitions` in terms of `source`.

An important design advice is to focus on thinking declaratively when programming in JastAdd. Think first about what attributes you would like the AST to have. Then, in defining these attributes, think of what other attributes that would be useful, in order to make your equations simple. This will lead to the addition of new attributes. In this tutorial, we have mostly worked in the other direction, in order to present simple mechanisms before more complex ones. For

a real situation, where you already know about the JastAdd mechanisms, you might have started out with the `reachable` attribute instead. In order to define it, it would have been useful to have the `successors` attribute. To define the `successors` attribute, you find that you need the `transitions` and `target` attributes, and so on.

The use of Java as the host language for writing equations is very powerful, allowing existing Java classes to be used for data types, and for connecting the attributed AST to imperatively implemented tools. At the same time, it is extremely important to understand the declarative semantics of the attribute grammars, and to watch out to not introduce any external side-effects in the equations. In particular, when dealing with composite values that are implemented using objects, it is very important to distinguish between their mutating and non-mutating operations, so that accessed attributes are not mutated by mistake.

As for normal object-oriented programming, naming is essential. Try to pick good descriptive names of both your AST classes and your attributes, so that the code you write is readable, and the APIs that the attributes produce will be simple and natural to use. For each attribute that you implement, you can write test cases that build up some example ASTs and test that the attributes get the intended values in different situations, so that you are confident that you have got your equations right.

JastAdd has been used for implementing both simple small languages and advanced programming languages. The implementation of our extensible Java compiler, JastAddJ, has been driving the development of JastAdd, and has motivated the introduction of many of the different mechanisms and made it possible to benchmark them on large programs [EH04, MH07, MEH09, NNEHM09]. Other advanced languages are being implemented as well, most notably an ongoing open-source implementation of the language Modelica which is used for describing physical models using differential equations [Mod10, JMo09, ÅEH10]. For more information about JastAdd, see `jastadd.org`.

Acknowledgments. The JastAdd system was jointly developed with my former PhD students Torbjörn Ekman and Eva Magnusson. For this tutorial I am particularly grateful to Torbjörn for joint work on constructing the state machine example. Thanks also to the students Nicolas Mora and Philip Nilsson whose comments helped me improve the readability of the tutorial, and to the anonymous reviewers who gave very detailed and valuable feedback on earlier drafts of this paper.

References

[ÅEH10] Åkesson, J., Ekman, T., Hedin, G.: Implementation of a Modelica compiler using JastAdd attribute grammars. Science of Computer Programming 73(1-2), 21–38 (2010)

[Boy96] Boyland, J.T.: Descriptional Composition of Compiler Components. PhD thesis, University of California, Berkeley, Available as technical report UCB//CSD-96-916 (September 1996)

[Boy05] Boyland, J.T.: Remote attribute grammars. J. ACM 52(4), 627–687 (2005)

[CK94] Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20(6), 476–493 (1994)

[DJL88] Deransart, P., Jourdan, M., Lorho, B.: Attribute Grammars: Definitions, Systems, and Bibliography. LNCS, vol. 323. Springer, Heidelberg (1988)

[DSST86] Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. In: Proceedings of the 18th Annual ACM Symposium on Theory of Computing, STOC 1986, pp. 109–121. ACM, New York (1986)

[EH04] Ekman, T., Hedin, G.: Rewritable Reference Attributed Grammars. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 147–171. Springer, Heidelberg (2004)

[EH06] Ekman, T., Hedin, G.: Modular name analysis for java using JastAdd. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 422–436. Springer, Heidelberg (2006)

[EH07a] Ekman, T., Hedin, G.: Pluggable checking and inferencing of non-null types for Java. Proceedings of TOOLS Europe 2007, Journal of Object Technology 6(9), 455–475 (2007)

[EH07b] Ekman, T., Hedin, G.: The Jastadd Extensible Java Compiler. In: OOPSLA 2007, pp. 1–18. ACM, New York (2007)

[Ekm06] Ekman, T.: Extensible Compiler Construction. PhD thesis, Lund University, Sweden (June 2006)

[Far86] Farrow, R.: Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In: Proceedings of the SIGPLAN Symposium on Compiler Construction, pp. 85–98. ACM Press, New York (1986)

[Hed94] Hedin, G.: An Overview of Door Attribute Grammars. In: Fritzson, P.A. (ed.) CC 1994. LNCS, vol. 786, pp. 31–51. Springer, Heidelberg (1994)

[Hed00] Hedin, G.: Reference Attributed Grammars. Informatica (Slovenia) 24(3), 301–317 (2000)

[JMo09] JModelica.org, Modelon AB (2009), `http://www.jmodelica.org`

[Jou84] Jourdan, M.: An optimal-time recursive evaluator for attribute grammars. In: Paul, M., Robinet, B. (eds.) Programming 1984. LNCS, vol. 167, pp. 167–178. Springer, Heidelberg (1984)

[Kas80] Kastens, U.: Ordered attribute grammars. Acta Informatica 13(3), 229–256 (1980); See also: Bericht 7/78, Institut für Informatik II, University Karlsruhe (1978)

[KHH+01] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–355. Springer, Heidelberg (2001)

[KHZ82] Kastens, U., Hutt, B., Zimmermann, E.: GAG: A Practical Compiler Generator. LNCS, vol. 141. Springer, Heidelberg (1982)

[Knu68] Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2(2), 127–145 (1968); Correction: Mathematical Systems Theory 5(1), 95–96 (1971)

[KSV09] Kats, L.C.L., Sloane, A.M., Visser, E.: Decorated attribute grammars: Attribute evaluation meets strategic programming. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 142–157. Springer, Heidelberg (2009)

[MEH09] Magnusson, E., Ekman, T., Hedin, G.: Demand-driven evaluation of collection attributes. Automated Software Engineering 16(2), 291–322 (2009)

[MH07] Magnusson, E., Hedin, G.: Circular Reference Attributed Grammars - Their Evaluation and Applications. Science of Computer Programming 68(1), 21–37 (2007)

[Mod10] The Modelica Association (2010), `http://www.modelica.org`

[NNEHM09] Nilsson-Nyman, E., Ekman, T., Hedin, G., Magnusson, E.: Declarative intraprocedural flow analysis of Java source code. Electronic Notes in Theoretical Computer Science 238(5), 155–171 (2009)

[Paa95] Paakki, J.: Attribute grammar paradigms - a high-level methodology in language implementation. ACM Computing Surveys 27(2), 196–255 (1995)

[PH97] Poetzsch-Heffter, A.: Prototyping realistic programming languages based on formal specifications. Acta Informatica 34, 737–772 (1997)

[RT84] Reps, T., Teitelbaum, T.: The Synthesizer Generator. In: ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments, pp. 42–48. ACM Press, Pittsburgh (April 1984)

[SKV09] Sloane, A.M., Kats, L.C.L., Visser, E.: A pure object-oriented embedding of attribute grammars. In: Ekman, T., Vinju, J. (eds.) Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications, LDTA 2009 (2009)

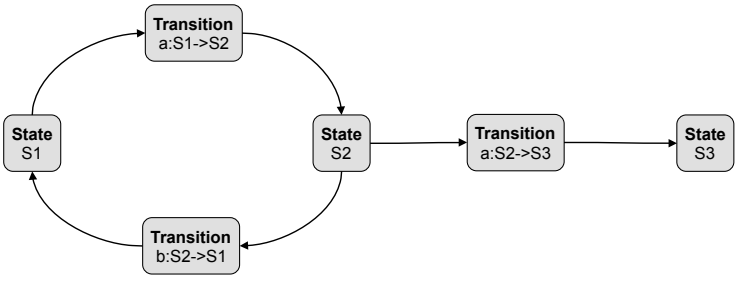
[UDP+82] Uhl, J., Drossopoulou, S., Persch, G., Goos, G., Dausmann, M., Winterstein, G., Kirchgässner, W.: An Attribute Grammar for the Semantic Analysis of ADA. LNCS, vol. 139. Springer, Heidelberg (1982)

[VSK89] Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher order attribute grammars. In: Proceedings of PLDI 1989, pp. 131–145. ACM Press, New York (1989)

[WBGK10] Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: An extensible attribute grammar system. Science of Computer Programming 75(1-2), 39–54 (2010)

A Solutions to Exercices

Exercise 1



Exercise 2

Here is an aspect defining a method `printInfoAboutCycles` for `StateMachine`:

```
aspect PrintInfoAboutCycles {
  public void StateMachine.printInfoAboutCycles() {
    for (Declaration d : getDeclarationList()) {
      d.printInfoAboutCycles();
    }
  }

  public void Declaration.printInfoAboutCycles() {}

  public void State.printInfoAboutCycles() {
    System.out.print("State "+getLabel()+" is ");
    if (!reachable().contains(this)) {
      System.out.print("not ");
    }
    System.out.println("on a cycle.");
  }
}
```

The main program parses an inputfile, then calls the `printInfoAboutCycles` method:

```
package exampleprogs;
import AST.*;
import java.io.*;

public class Compiler {
  public static void main(String[] args) {
    String filename = getFilename(args);
```

```
    // Construct the AST
    StateMachine m = parse(filename);

    // Print info about what states are on cycles
    m.printInfoAboutCycles();
  }

  public static String getFilename(String[] args) { ... }

  public static StateMachine parse(String filename) { ... }
}
```

Running the "compiler" on the input program of Fig. 3, gives the following output:

```
State S1 is on a cycle.
State S2 is on a cycle.
State S3 is not on a cycle.
```

Exercise 3

```
C.v = 18
C.i = 7
E.s = 26
E.i = 18
F.t = 3
G.u = 5
```

Exercise 4

`B`, `D`, `F`, and `G`.

Exercise 5

There are many possible algorithms for computing attribute values in an AST. Here are some alternatives:

Dynamic, with explicit depedency graph. Add dependency edges between all attributes in an AST according to the equations. For example, for an equation $a = f(b, c)$, the two edges (b, a) and (c, a) are added. Then run all the equations as assignments in topological order, starting with equations with no incoming edges. This kind of algorithm is called a *dynamic* algorithm because we use the dependencies of an actual AST, rather than only *static* dependencies that we can derive from the grammar.

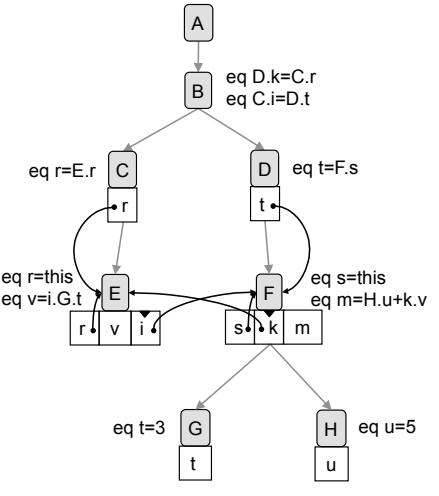
Static. Compute a conservative approximation of a dependency graph, based on the attribute grammar alone, so that the graph holds for any possible AST. Then compute a scheduling for in what order to evaluate the attributes based on this graph. This kind of algorithm is called a *static* algorithm, since it

only takes the grammar into account, and not the actual AST. It might be more efficient than the dynamic algorithm since the actual dependencies in an AST do not need to be analyzed. On the other hand, it will be less general because of the approximation. There are many different algorithms that use this general approach. They differ in how they do the approximation, and thereby in how general they are.

Dynamic, with implicit dependency graph. Represent each attribute by a function, corresponding to the right-hand side of its defining equation. To evaluate an attribute, simply call its function. Recursive calls to other attributes will automatically make sure the attributes are evaluated in topological order. This algorithm is also dynamic, but does not require building an explicit dependency graph.

The JastAdd system uses this latter algorithm, see Section 3.8 for details. The other two alternatives are not powerful enough to handle arbitrary reference attributes.

Exercise 6

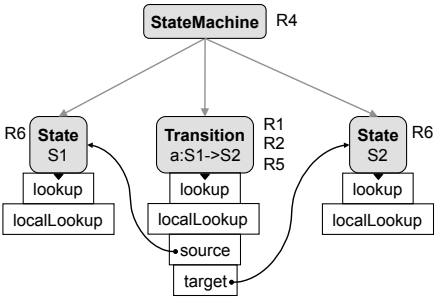


`E` and `F` are on a cycle via their attributes `i` and `k`.
Values of non-reference attributes:

```
E.v = 3
F.m = 8
G.t = 3
H.u = 5
```

Examples of non-local dependencies: the value of `E.v` depends directly on the value of `G.t`, and `F.m` depends directly on `E.v`.

Exercise 7



Exercise 8

```
syn boolean State.alreadyDeclared() =
    lookup(this.getLabel()) != this;
```

Exercise 9

A state is multiply declared if it either is declared already, or if it has a later namesake. To find out if it has a later namesake, we define a new attribute `lookupForward` that only traverses declarations *after* the state. Note that the equation for this attribute makes use of the argument `i` to start traversing at the appropriate position in the list.

```
syn boolean State.multiplyDeclared() =
    alreadyDeclared() || hasLaterNamesake();

syn boolean State.hasLaterNamesake() =
    lookupForward(getLabel()) != null;

inh State Declaration.lookupForward(String label);

eq StateMachine.getDeclaration(int i).lookupForward(String label) {
  for (int k = i+1; k<getNumDeclaration(); k++) {
    Declaration d = getDeclaration(k);
    State match = d.localLookup(label);
    if (match != null) return match;
  }
  return null;
}
```

Exercise 10

```
syn Set<Transition> State.altTransitions() = transitionsOf(this);
inh Set<Transition> State.transitionsOf(State s);
```

```
eq StateMachine.getDeclaration(int i).transitionsOf(State s) {
  HashSet<Transition> result = new HashSet<Transition>();
  for (Declaration d : getDeclarationList()) {
    Transition t = d.transitionOf(s);
    if (t != null) result.add(t);
  }
  return result;
}

syn Transition Declaration.transitionOf(State s) = null;
eq Transition.transitionOf(State s) {
  if (source() == s)
    return this;
  else
    return null;
}
```

We see that the definition of `altTransitions` is more complex than that of `transitions`: two help attributes are needed: the inherited `transitionsOf` and the synthesized `transitionOf`. Furthermore, we see that the definition of `altTransitions` is more coupled in that it relies on both the existence of the `StateMachine` nodeclass, and on its child structure.

Exercise 11

```
coll Set<State> State.altSuccessors()
  [new HashSet<State>()] with add;

Transition contributes target()
  when target() != null && source() != null
  to State.altSuccessors()
  for source();
```

In this case, the definitions using ordinary attributes and collection attributes have about the same complexity and coupling.

Exercise 12

```
coll Set<State> State.predecessors()
  [new HashSet<State>()] with add;

State contributes this
  to State.predecessors()
  for each successors();
```

Exercise 13

To define the collection, we introduce a class `Counter` that works as a wrapper for Java integers. To give `Transitions` access to their enclosing state machine node,

in order to contribute the value 1 to the collection, we introduce an inherited attribute `theMachine`. Finally, the synthesized attribute `numberOfTransitions` simply accesses the value of the `Counter`.

```
syn int StateMachine.numberOfTransitions() =
    numberOfTransitionsColl().value();
coll Counter StateMachine.numberOfTransitionsColl()
     [new Counter()] with add;

Transition contributes 1
  to StateMachine.numberOfTransitionsColl()
  for theMachine();

inh StateMachine Declaration.theMachine();
eq  StateMachine.getDeclaration(int i).theMachine() = this;

class Counter {
  private int value;
  public Counter() { value = 0; }
  public void add(int value) { this.value += value; }
  public int value() { return value; }
}
```

Exercise 14

Here we have simply used a set of strings to represent the error messages. In addition to missing declarations of states, error messages are added for states that are declared more than once.

```
coll Set<String> StateMachine.errors()
     [new HashSet<String>()] with add;

State contributes getLabel()+" is already declared"
when alreadyDeclared()
to StateMachine.errors()
for theMachine();

Transition contributes "Missing declaration of "+getSourceLabel()
when source() == null
to StateMachine.errors()
for theMachine();

Transition contributes "Missing declaration of "+getTargetLabel()
when target() == null
to StateMachine.errors()
for theMachine();
```

Exercise 15

$$reachable_1 = \{S_2\} \cup reachable_2$$
$$reachable_2 = \{S_1, S_3\} \cup reachable_1 \cup reachable_2$$
$$reachable_3 = \emptyset$$

Exercise 16

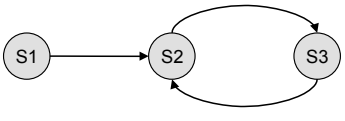
The least (and desired) solution is

$$reachable_1 = \{S_1, S_2, S_3\}$$
$$reachable_2 = \{S_1, S_2, S_3\}$$
$$reachable_3 = \emptyset$$

There are no additional solutions since the attributes that are circular ($reachable_1$ and $reachable_2$) have the top value in the lattice (the set of all states).

Exercise 17

This state machine has more than one solution for *reachable*.



The equation system is:

$$reachable_1 = \{S_2\} \cup reachable_2$$
$$reachable_2 = \{S_3\} \cup reachable_3$$
$$reachable_3 = \{S_2\} \cup reachable_2$$

The least (and desired) solution is:

$$reachable_1 = \{S_2, S_3\}$$
$$reachable_2 = \{S_2, S_3\}$$
$$reachable_3 = \{S_2, S_3\}$$

An additional (and uninteresting) solution also includes S_1:

$$reachable_1 = \{S_1, S_2, S_3\}$$
$$reachable_2 = \{S_1, S_2, S_3\}$$
$$reachable_3 = \{S_1, S_2, S_3\}$$

Exercise 18

```
coll Set<State> State.altReachable() circular
  [new HashSet<State>()] with addAll;

State contributes union(asSet(this),altReachable())
  to State.altReachable()
  for each predecessors();
```

In the above solution we have made use of two auxiliary functions: `asSet` and `union`. It would have been nice if these functions had already been part of the Java Set interface, but since they are not, we define them as functions in `ASTNode` as shown below, making them available to all AST nodes. (The class `ASTNode` is a superclass of all node classes.) A nicer solution can be achieved by designing new alternative Java classes and interfaces for sets.

```
Set<State> ASTNode.asSet(State o) {
  HashSet<State> result = new HashSet<State>();
  result.add(o);
  return result;
}

Set<State> ASTNode.union(Set<State> s1, Set<State> s2) {
  HashSet<State> result = new HashSet<State>();
  for (State s: s1) result.add(s);
  for (State s: s2) result.add(s);
  return result;
}
```

Model Driven Language Engineering with Kermeta

Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey

INRIA & University of Rennes1
Campus Universitaire de Beaulieu
35042 Rennes CEDEX, France

Abstract. In many domains such as telecom, aerospace and automotive industries, engineers rely on Domain Specific Modeling Languages (DSML) to solve the complex issues of engineering safety critical software. Traditional Language Engineering starts with the grammar of a language to produce a variety of tools for processing programs expressed in this language. Recently however, many new languages tend to be first defined through metamodels, i.e. models describing their abstract syntax. Relying on well tooled standards such as E-MOF, this approach makes it possible to readily benefit from a set of tools such as reflexive editors, or XML serialization of models. This article aims at showing how Model Driven Engineering can easily complement these off-the-shelf tools to obtain a complete environment for such a language, including interpreter, compiler, pretty-printer and customizable editors. We illustrate the conceptual simplicity and elegance of this approach using the running example of the well known LOGO programming language, developed within the Kermeta environment.

1 Introduction

In many domains such as telecom, aerospace and automotive industries [21], engineers rely on Domain Specific Modeling Languages (DSML) to solve the complex issues of engineering safety critical software at the right level of abstraction. These DSMLs indeed define modeling constructs that are tailored to the specific needs of a particular domain. When such a new DSML is needed, it is now often first defined through meta-models, i.e. models describing its abstract syntax [14] when traditional language engineering would have started with the grammar of the language. Relying on well tooled standards such as E-MOF, the meta-modeling approach makes it possible to readily benefit from a set of tools such as reflexive editors, or XML serialization of models. More importantly, having such a tool supported *de facto* standard for defining models and meta-models paves the way towards a rich ecosystem of interoperable tools working seamlessly with these models and meta-models.

Combining this Model Driven approach with a traditional grammar based one has however produced mixed results in terms of the complexity of the overall approach. Several groups around the world are thus investigating the idea of a

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 201–221, 2011.

new Language Engineering completely based on models [22], that we call Model Driven Language Engineering (MDLE).

In this paper we present one of these approaches, based on the Kernel Meta-Modeling environment Kermeta [16,7]. We start in Section 2 by giving a quick overview of executable meta-modeling, and then focusing on Kermeta, seen both as an aspect-oriented programming language as well as an integration platform for heterogeneous meta-modeling. We then recall in Section 3 how to model the abstract syntax of a language in E-MOF, allowing for a direct implementation of its meta-model in the Eclipse Modeling Framework (EMF). We then show how to weave both the static and dynamic semantics of the language into the meta-model using Kermeta to get an interpreter for the language. Then we address compilation, which is just a special case of model transformation to a platform specific model [17,3]. We illustrate the conceptual simplicity and elegance of this approach using the running example of the well known Logo programming language, for which a complete programming environment is concretely outlined in this article, from the Logo meta-model to simulation to code generation for the Lego Mindstorm platform and execution of a Logo program by a Mindstorm turtle.

2 Executable Meta-modeling

2.1 Introduction

Modeling is not just about expressing a solution at a higher abstraction level than code. This limited view on modeling has been useful in the past (assembly languages abstracting away from machine code, 3GL abstracting over assembly languages, etc.) and it is still useful today to get e.g.; a holistic view on a large C++ program. But modeling goes well beyond that.

In engineering, one wants to break down a complex system into as many models as needed in order to address all the relevant concerns in such a way that they become understandable enough. These models may be expressed with a general purpose modeling language such as the UML [26], or with Domain Specific Modeling Languages (DSML) when it is more appropriate. Each of these models can be seen as the abstraction of an aspect of reality for handling a given concern. The provision of effective means for handling such concerns makes it possible to establish critical trade-offs early on in the software life cycle.

Models have been used for long as *descriptive* artifacts, which was already extremely useful. In many cases we want to go beyond that, i.e. we want to be able to perform computations on models, for example to simulate some behavior [16], or to generate code or tests out of them [19]. This requires that models are no longer informal, and that the language used to describe them has a well defined abstract syntax (called its meta-model) and semantics.

Relying on well tooled Eclipse standards such as E-MOF to describe these meta-models, we can readily benefit from a set of tools such as reflexive editors, or XML serialization of models, and also from a standard way of accessing models

from Java. The rest of this section introduces Kermeta, a Kernel Meta-Modeling language and environment, whose goal is to complement Eclipse off-the-shelf tools to obtain a complete environment for such DSMLs, including interpreters, compilers, pretty-printers and customizable editors.

2.2 Kermeta as a MOF Extension

Kermeta is a Model Driven Engineering platform for building rich development environments around meta-models using an aspect-oriented paradigm [16,10]. Kermeta has been designed to easily extend meta-models with many different concerns (such as static semantics, dynamic semantics, model transformations, connection to concrete syntax, etc.) expressed in heterogeneous languages. A meta-language such as the Meta Object Facility (MOF) standard [18] indeed already supports an object-oriented definition of meta-models in terms of packages, classes, properties and operation signatures, as well as model-specific constructions such as containments and associations between classes. MOF does not include however concepts for the definition of constraints or operational semantics (operations in MOF do jot contain bodies). Kermeta can thus be seen as an extension of MOF with an imperative action language for specifying constraints and operation bodies at the meta-model level.

The action language of Kermeta is especially designed to process models. It is imperative and includes classical control structures such as blocks, conditional and loops. Since the MOF specifies object-oriented structures (classes, properties and operations), Kermeta implements traditional object-oriented mechanisms for multiple inheritance and behavior redefinition with a late binding semantics (to avoid multiple inheritance conflicts a simple behaviors selection mechanism is available in Kermeta). Like most modern object-oriented languages, Kermeta is statically typed, with generics and also provides reflection and an exception handling mechanism.

In addition to object-oriented structures, the MOF contains model-specific constructions such as containment and associations. These elements require a specific semantics of the action languages in order to maintain the integrity of associations and containment relations. For example, in Kermeta, the assignment of a property must handle the other end of the association if the property is part of an association and the object containers if the property is a composition.

Kermeta expressions are very similar to Object Constraint Language (OCL) expressions. In particular, Kermeta includes lexical closures similar to OCL iterators on collections such as *each, collect, select* or *detect*. The standard framework of Kermeta also includes all the operations defined in the OCL standard framework. This alignment between Kermeta and OCL allows OCL constraints to be directly imported and evaluated in Kermeta. Pre-conditions and post-conditions can be defined for operations and invariants can be defined for classes. The Kermeta virtual machine has a specific execution mode, which monitors these contracts and reports any violation.

2.3 Kermeta as an Aspect-Oriented Integration Platform

Since Kermeta is an extension of MOF, a MOF meta-model can conversely be seen as a valid Kermeta program that just declares packages, classes and so on but *does* nothing. Kermeta can then be used to *breath life* into this meta-model by incrementally introducing aspects for handling concerns of static semantics, dynamic semantics, or model transformations [17].

One of the key features of Kermeta is the static composition operator "*require*", which allows extending an existing meta-model with new elements such as properties, operations, constraints or classes. This operator allows defining these various aspects in separate units and integrating them automatically into the meta-model. The composition is done statically and the composed model is typed-checked to ensure the safe integration of all units. This mechanism makes it easy to reuse existing meta-models or to split meta-models into reusable pieces. It can be compared to the open class paradigm [4]. Consequently a meta-class that identifies a domain concept can be extended without editing the meta-model directly. Open classes in Kermeta are used to organize "cross-cutting" concerns separately from the meta-model to which they belong, a key feature of aspect-oriented programming [11]. With this mechanism, Kermeta can support the addition of new meta-class, new subclasses, new methods, new properties, new contracts to existing meta-model. The *require* mechanism also provides flexibility. For example, several operational semantics could be defined in separate units for a single meta-model and then alternatively composed depending on particular needs. This is the case for instance in the UML meta-model when several semantics variation points are defined.

Thank to this composition operator, Kermeta can remain a kernel platform to safely integrate all the concerns around a meta-model. As detailed in the previous paragraphs, meta-models can be expressed in MOF and constraints in OCL. Kermeta also allows importing Java classes in order to use services such as file input/output or network communications during a transformation or a simulation. These functionalities are not available in the Kermeta standard framework. Kermeta and its framework remain dedicated to model processing but provide an easy integration with other languages. This is very useful for instance to make models communicating with existing Java applications.

3 Building an Integrated Environement for the Logo Language

3.1 Meta-modeling Logo

To illustrate the approach proposed in this paper, we use the example of the Logo language. This example was chosen because Logo is a simple yet real (i.e. Turing-complete) programming language, originally created for educational purposes. Its most popular application is *turtle graphics*: the program is used to direct a virtual turtle on a board and make it draw geometric figures when its pen

```
1  # definition of the square procedure
2  TO square :size
3    REPEAT 4 [
4      FORWARD :size
5      RIGHT 90
6    ]
7  END
8
9  # clear screen
10 CLEAR
11
12 # draw a square
13 PENDOWN
14 square(50)
15 PENUP
```

Fig. 1. Logo square program

is down[1]. Figure 1 presents a sample Logo program which draws a square. In this paper we propose to build a complete Logo environment using model-driven engineering techniques.

The first task in the model driven construction of a language is the definition of its abstract syntax. The abstract syntax captures the concepts of the language (these are primitive instructions, expressions, control structures, procedure definitions, etc.) and the relations among them (e.g. an expression is either a constant or a binary expression, that itself contains two expressions). In our approach the abstract syntax is defined using a meta-model.

Figure 2 presents the meta-model for the abstract syntax of the Logo language. The Logo meta-model includes:

- Primitive statements (*Forward*, *Back*, *Left*, *Right*, *PenUp* and *PenDown*). These statement allows moving and turning the Logo turtle and controlling its pen.
- Arithmetic Expressions (*Constant*, *BinaryExp* and its sub-classes). In our version of Logo, constants are integers and all operators only deal with integers.
- Procedures (*ProcDeclaration*, *ProcCall*, *Parameter* and *ParameterCall*) allow defining and calling functions with parameters (note that recursion is supported in Logo).
- Control Structures (*Block*, *If*, *Repeat* and *While*). Classical sequence, conditional and loops for an imperative language.

In practice the Logo meta-model can be defined within the Eclipse Modeling Framework (EMF). EMF is a meta-modeling environment built on top of the

[1] A complete history of the Logo language and many code samples can be found on wikipedia (http://en.wikipedia.org/wiki/Logo_(programming_language))

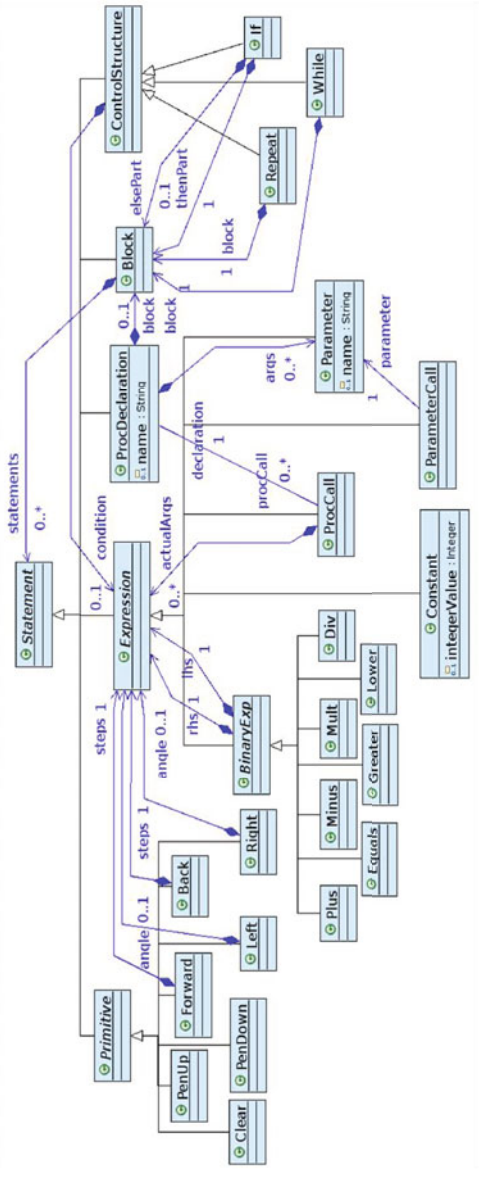


Fig. 2. Logo Abstract Syntax

Eclipse platform and based on the Essential-MOF standard. Within Eclipse several graphical editors can be used to define such meta-models. Once the meta-model is defined, the EMF automatically provides editors and serialization capabilities for the meta-model. The editor allows creating instances of the classes of the meta-model and saving these instances models in the XMI standard format.

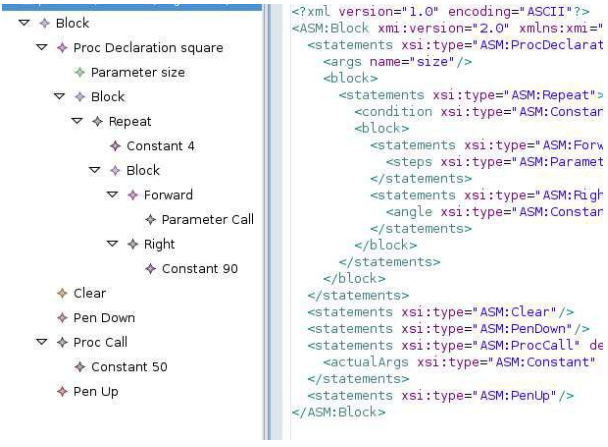


Fig. 3. Logo square program in the generated editor and serialized in XMI

As soon as the meta-model of Fig. 2 has been defined, it is possible to instantiate it using the generated editor in order to write Logo programs. Figure 3 presents a screen-shot of the generated editor with the square program presented previously. The program was defined in the tree editor and the right part of the figure shows how the logo program was serialized.

3.2 Weaving Static Semantics

The Object Constraint Language. A meta-model can be seen as the definition of the set of allowed configurations for a set of objects representing a domain. All structures are represented as classes, relations and structural properties. In MDLE, a meta-model defines a set of valid programs. However, some constraints (*formulas* to the logician, *Boolean expressions* to the programmer) cannot directly be expressed using EMOF. For example there is no easy way to express that formal parameter names should be unique in a given procedure declaration, or that in a valid Logo program the number of actual arguments in a procedure call should be the same as the number of formal arguments in the declaration. This kind of constraints forms part of what is often called the static semantics of the language.

In Model-Driven Engineering, the Object Constraint Language (OCL) [20] is often used to provide a simple first order logic for the expression of the static semantics of a meta-model. OCL is a declarative language initially developed at IBM for describing constraints on UML models. It is a simple text language that provides constraints and object query expressions on any Meta-Object Facility model or meta-model that cannot easily be expressed by diagrammatic notation. OCL language statements are constructed using the following elements:

1. a context that defines the limited situation in which the statement is valid
2. a property that represents some characteristics of the context (e.g., if the context is a class, a property might be an attribute)
3. an operation (e.g., arithmetic, set-oriented) that manipulates or qualifies a property, and
4. keywords (e.g., if, then, else, and, or, not, implies) that are used to specify conditional expressions.

Expressing the Logo Static Semantics in OCL. The Logo meta-model defined on figure 2 only defines the structure of a Logo program. To define the sub-set of programs which are valid with respect to Logo semantics a set of constraints has to be attached to the abstract syntax. Figure 4 presents the OCL listing of two constraints attached to the Logo meta-model. The first one is an invariant for class *ProcCall* that ensures that any call to a procedure has the same number of actual arguments as the number of formal arguments in the procedure declaration. The second invariant is attached to class *ProcDeclaration* and ensures that the names of the formal parameters of the procedure are unique.

```
1  package kmLogo::ASM
2
3  context ProcCall
4  inv same_number_of_formals_and_actuals :
5    actualArgs->size() = declaration.args->size()
6
7  context ProcDeclaration
8  inv unique_names_for_formal_arguments :
9    args->forAll ( a1, a2 | a1.name = a2.name implies a1 =
         a2 )
10
11 endpackage
```

Fig. 4. OCL constraint on the Logo meta-model

Weaving the Logo Static Semantics into its Meta-Model. In the Kermeta environment, the OCL constraints are woven directly into the meta-model and can be checked for any model which conforms to the Logo meta-model. In practice, once the designer has created a meta-model with the E-core formalism in the Eclipse Modeling Framework (called e.g. `ASMLogo.ecore`), she can import it into Kermeta (see line 2 in Figure 5) using the *require* instruction as described in Section 2. Suppose now that the Logo static semantics (of Fig. 4) is located in a file called `StaticSemantics.ocl`. Then the same *require* instruction can be used in Kermeta to import the Logo static semantics and weave it into the Logo meta-model (see line 3 in Figure 5).

```
1  package kmLogo;
2  require "ASMLogo.ecore"
3  require "StaticSemantics.ocl"
4  [...]
5  class Main {
6    operation Main(): Void is do
7    // Load a Logo program and check constraints on it
8    // then run it
9    end
10 end
```

Fig. 5. Weaving Static Semantics into the Logo Meta-Model

The integration of OCL into Kermeta relies onto two features:

- First, as presented in Section 3, Kermeta already supports a native constraint system made of invariants, pre and post conditions which makes it possible to work within a Design-by-Contracts methodology.
- Secondly, the support for the OCL concrete syntax is implemented with a model transformation from the AST of OCL to the AST of Kermeta. This transformation has been written in Kermeta. The result model of this transformation is automatically associated to the meta-model implicated, using the built-in static introduction of Kermeta.

Kermeta allows the user to choose when his constraints should be checked. That can be done class by class or at the entire model level with primitive *checkInvariant* on class or *checkAllInvariants* on the root element of the meta-model. The operation constraints (*pre*, *post*) are optionally checked depending on the type of "Run" chosen from the Eclipse menu: normal run or run with constraint checking.

So, at this stage the meta-model allows defining Logo programs with a model editor provided by the EMF and this model can be validated with respect to Logo static semantics within the Kermeta environment. For instance if we modify the Logo program of Fig. 1 by calling `square(50,10)` instead of `square(50)`, and if we load it into Kermeta, then by calling *checkAllInvariants* we get the expected error message that

```
Invariant same_number_of_formals_and_actuals
  has been violated for: square(50,10)
```

One point of interest is that this implementation extends the expressiveness of OCL. OCL already offers the possibility to call operations or derived properties declared in the meta-model. Kermeta allows the designer to specify the operational semantic of these methods or these properties. Then, using the OCL implementation in Kermeta, it is possible to express any expression based on the first-order logic and extend it with the imperative operations provided by Kermeta. Designer must of course still guarantee that these operations are free from side-effects on the abstract state of the models.

3.3 Weaving Dynamic Semantics to Get an Interpreter

The next step in the construction of a Logo environment is to define Logo operational semantics and to build an interpreter. In our approach this is done in two steps. The first one is to define the runtime model to support the execution of Logo programs, i.e. the Logo *virtual machine*. The second one is to define a mapping between the abstract syntax of Logo and this virtual machine. This is going to be implemented as a set of *eval* functions woven into the relevant constructs of the Logo meta-model.

Logo runtime model. As discussed earlier, the most popular runtime model for Logo is a turtle which can draw segments with a pen. As for the language abstract syntax, the structure of the runtime model can be defined by a meta-model. The advantage of this approach is that the state of the running program is then also a model. Like for any model, all the tools available in a framework like EMF can then readily be used in order to observe, serialize, load or edit the state of the runtime.

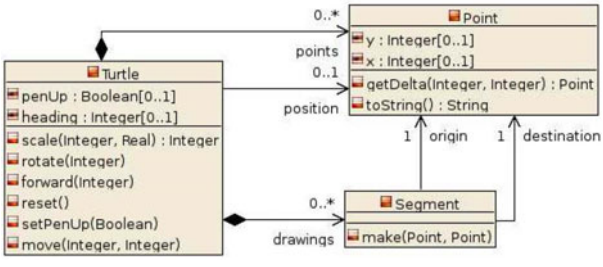


Fig. 6. Logo runtime meta-model

Figure 6 presents a diagram of the Logo virtual machine meta-model. The meta-model only defines three classes: *Turtle*, *Point* and *Segment*. The state of the running program is modeled as a single instance of class *Turtle* which has a position (which is a *Point*), a heading (which is given in degrees) and a Boolean to represent whether the pen is up or down. The Turtle stores the segments which were drawn during the execution of the program. In practice the meta-model was defined without operation using EMF tools. The operations, implemented in Kermeta, have been later woven into the meta-model to provide an object-oriented definition of the Logo virtual machine. Figure 7 presents an excerpt of the Kermeta listing. It adds three operations to the class *Turtle* of the meta-model.

Operational semantics. We are now going to define the operational semantics for each constructs of the abstract syntax. The idea is again to weave operations implemented in Kermeta directly into the meta-model in such a way that each type of statement would contain an *eval* operation performing the appropriate actions on the underlying runtime model. To do that, a context is provided as a

```
1  package kmLogo;
2
3  require "VMLogo.ecore"
4  [...]
5  package VM {
6    aspect class Turtle {
7      operation setPenUp(b : Boolean) is do
8        penUp := b
9      end
10     operation rotate(angle : Integer)  is do
11       heading := (heading + angle).mod(360)
12     end
13     operation forward(steps : Integer)  is do
14       var radian : Real init math.toRadians(heading.
             toReal)
15       move(scale(steps,math.sin(radian)), scale(steps,
             math.cos(radian)))
16     end
17     [...]
18   }
19 }
```

Fig. 7. Runtime model operations in Kermeta

parameter of the *eval* operation. This context contains an instance of the Turtle class of the runtime meta-model and a stack to handle procedure calls. Figure 8 presents how the operation *eval* are woven into the abstract syntax of Logo. An abstract operation *eval* is defined on class *Statement* and implemented in every sub-class to handle the execution of all constructions.

For simple cases such as the *PenDown* instruction, the mapping to the virtual machine is straightforward: it only boils down to calling the relevant VM instruction, i.e. `context.turtle.setPenUp(false)` (see line 36 of Fig. 8).

For more complex cases such as the *Plus* instruction, there are two possible choices. The first one, illustrated on lines 9–13 of Fig. 8, makes the assumption that the semantics of the Logo *Plus* can be directly mapped to the semantics of "+" in Kermeta. The interest of this first solution is that it provides a quick and straightforward way of defining the semantics of that kind of operators. If however the semantics we want for the Logo *Plus* is not the one that is built-in Kermeta for whatever reason (e.g. we want it to handle 8-bits integers only), we can define the wanted *Plus* operation semantics in the Logo Virtual Machine (still using Kermeta of course) and change the *eval* method of lines 9–13 so that it first calls *eval* on the left hand side, push the result on the VM stack, then calls *eval* on the right hand side, again push the result on the VM stack, and finally call the *Plus* operation on the VM.

```
package kmLogo;
require "ASMLogo.ecore"
require "LogoVMSemantics.kmt"
[...]
package ASM {
  aspect class Statement {
    operation eval(context : Context) : Integer is
        abstract
  }
  aspect class Plus {
    method eval(context : Context) : Integer is do
      result := lhs.eval(context) + rhs.eval(context)
    end
  }
  aspect class Greater {
    method eval(context : Context) : Integer is do
      result := if lhs.eval(context) > rhs.eval(context)
          then 1 else 0 end
    end
  }
  aspect class If {
    method eval(context : Context) : Integer is do
      if condition.eval(context) != 0 then
        result := thenPart.eval(context)
      else
        result := elsePart.eval(context)
      end
    end
  }
  aspect class Forward {
    method eval(context : Context) : Integer is do
      context.turtle.forward(steps.eval(context))
      result := void
    end
  }
  aspect class PenDown {
    method eval(context : Context) : Integer is do
      context.turtle.setPenUp(false)
      result := void
    end
  }
  [...]
}
```

Fig. 8. Logo operational semantics

```
1  package kmLogo;
2  require "ASMLogo.ecore"
3  require "StaticSemantics.ocl"
4  require "LogoVMSemantics.kmt"
5  require "OperationalSemantics.kmt"
6  [...]
7  class Main {
8    operation Main(): Void is do
9      var rep : EMFRepository init EMFRepository.new
10     var logoProgram : ASMLogo::Block
11     // load logoProgram from its XMI file
12     logoProgram ?= rep.getResource("Square.xmi").one
13     // Create a new Context containing the Logo VM
14     var context : LogoVMSemantics::Context init
           LogoVMSemantics::Context.new
15     // now execute the logoProgram
16     logoProgram.eval(context)
17   end
18 end
```

Fig. 9. Getting an Interpreter

Getting an Interpreter. Once the operational semantics for Logo has been defined as described above, getting an interpreter is pretty straightforward: we first have to import each relevant aspect to be woven into the Logo meta-model (using *require* instructions, see lines 2–5 in Fig. 9). We then need to load the Logo program into Kermeta (see lines 9–12 in Fig. 9), instantiate a *Context* (that contains the Logo VM) and then call *eval(Context)* on the root element of the Logo program.

Loading the *Square* program of Fig. 1 and executing it this way will change the state of the model of the Logo VM: during the execution, four new *Segments* will be added to the *Turtle*, and its position and heading will change. Obviously, we would like to see this execution graphically on the screen. The solution is quite easy: we just need to put an Observer on the Logo VM to graphically display the resulting figure in a Java widget. The Observer is implemented in Kermeta and calls relevant Java methods to notify the screen that something has changed.

3.4 Compilation as a Kind of Model Transformation

In this section we are going to outline how to build a compiler for our Logo language. The idea is to map a Logo program to the API offered by the Lego Mindstroms so that our Logo programs can actually be used to drive small robots mimicking Logo turtles. These *Robot-Turtles* are built with Lego Mindstroms and feature two motors for controlling wheels and a third one for controlling a pen (see Fig. 10).

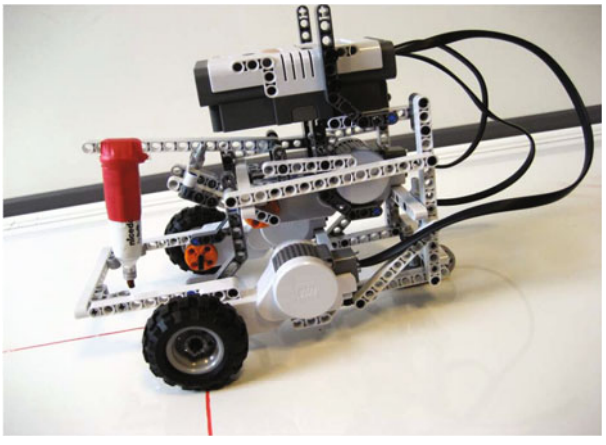


Fig. 10. A Lego mindstorm robot-turtles

A simple programming language for Lego Mindstroms is NXC (standing for Not eXactly C). So building a Logo compiler for Lego Mindstroms boils down to write a translator from Logo to NXC. The problem is thus much related to the Model Driven Architecture (MDA) context as promoted by the OMG, where a Logo program would play the role of a Platform Independent Model (PIM) while the NXC program would play the role of a Platform Specific Model (PSM). With this interpretation, the compilation we need is simply a kind of model transformation.

We can implement this model transformation either using Model-to-Model Transformations or Model-to-Text Transformations:

Model-to-Text Transformations are very useful for generating code, XML, HTML, or other documentation in a straightforward way, when the only thing that is needed is actually a syntactic level transcoding (e.g. Pretty-Printing). Then we can resort on either:

- Visitor-Based Approaches, where some visitor mechanisms are used to traverse the internal representation of a model and directly write code to a text stream.
- Template-Based Approaches, based on the target text containing slices of meta-code to access information from the source and to perform text selection and iterative expansion. The structure of a template resembles closely the text to be generated. Textual templates are independent of the target language and simplify the generation of any textual artifacts.

Model-to-Model Transformations would be used to handle more complex, semantic driven transformations.

For example if complex, multi-pass transformations would have been needed to translate Logo to NXC, it could have been interesting to have an explicit

meta-model of NXC, properly implement the transformation between the Logo meta-model and the NXC one, and finally call a pretty-printer to output the NXC source code.

In our case however the translation is quite simple, so we can for example directly implement a visitor-based approach. In practice, we are once again going to use the aspect weaving mechanism of Kermeta simplify the introduction of the Visitor pattern. Instead of using the pair of methods *accept* and *visit*, where each *accept* method located in classes of the Logo meta-model would call back the relevant *visit* method of the visitor, we can directly weave a *compile()* method into each of these Logo meta-model classes (see Fig. 11).

```
1  package kmLogo;
2
3  require "ASMLogo.ecore"
4  [...]
5  package ASMLogo {
6    aspect class PenUp {
7      compile (ctx: Context) {
8      [...]
9      }
10   }
11
12   aspect class Clear {
13     compile (ctx: Context) {
14     [...]
15     }
16   }
17 [...]
18 }
```

Fig. 11. The Logo Compilation Aspect in Kermeta

Integrating this compilation aspect into our development environment for Logo is done as usual, i.e. by *requiring* it into the main Kermeta program (see Fig. 12).

3.5 Model to Model Transformation

For the Logo compiler described above to work properly, we have to assume though that all Logo function declarations are performed at the outermost block level, because NXC does not support nested function declarations. Since nothing in our Logo meta-model prevents the Logo user to declare nested functions, we need to either add an OCL constraint to the Well-Formedness Rules of the language, or we need to do some pre-processing before the actual compilation

```
1  package kmLogo;
2  require "ASMLogo.ecore"
3  require "StaticSemantics.ocl"
4  require "Compiler.kmt"
5  [...]
6  class Main {
7    operation Main(): Void is do
8      var rep : EMFRepository init EMFRepository.new
9      var logoProgram : ASMLogo::Block
10     // load logoProgram from its XMI file
11     logoProgram ?= rep.getResource("Square.xmi").one
12     // Create a new Context for storing global data
           during the compilation
13     var context : Context init Context.new
14     // now compile the logoProgram to NXC
15     logoProgram.compile(context)
16   end
17 end
```

Fig. 12. Getting a Compiler

step. For the sake of illustrating Kermeta capabilities with respect to Model to Model Transformations, we are going to choose the later solution.

We thus need a new aspect in our development environment, that we call the *local-to-global* aspect (See Listing 13) by reference to an example taken from the TXL [5] tutorial. We are using a very simple OO design that declares an empty method *local2global* (taking as parameter the root block of a given Logo program) in the topmost class of the Logo meta-model hierarchy, *Statement*. We are then going to redefine it in relevant meta-model classes, such as *ProcDeclaration* where we have to move the current declaration to the root block and recursively call *local2global* on its block (containing the function body). Then in the class *Block*, the *local2global* method only has to iterate through each instruction and recursively call itself.

Note that if we also allow *ProcDeclaration* inside control structure such as *Repeat* or *If*, we would also need to add a *local2global* method in these classes to visit their block (*thenPart* and *elsePart* in the case of the *If* statement).

Once again this *local2global* concern is implemented in a modular way in Kermeta, and can easily be added or removed from the Logo programming environment without any impact on the rest of the software. Further, new instructions could be added to Logo (i.e. by extending its meta-model with new classes) without much impact on the *local2global* concern as long as these instructions do not contain any block structure. This loose coupling is a good illustration of Kermeta advanced modularity features, allowing both easier parallel development and maintenance of a DSML environment.

```
1  package kmLogo;
2
3  require "ASMLogo.ecore"
4  [...]
5  package ASMLogo {
6  aspect class Statement
7    method local2global(rootBlock: Block) is do
8    end
9  end
10 aspect class ProcDeclaration
11   method local2global(rootBlock: Block) is do
12      rootBlock.add(self)
13      block.local2global(rootBlock)
14   end
15 end
16 aspect class Block
17   method local2global(rootBlock: Block) is do
18       statements.each(i| i.local2global(rootBlock))
19    end
20 end
21 }
```

Fig. 13. The Logo *local-to-global* Aspect in Kermeta

4 Discussion

4.1 Separation of Concerns for Language Engineering

From an architectural point of view, Kermeta allows the language designer to keep his concerns separated. Designers of meta-models will typically work with several artifacts: the structure is expressed in the *Ecore* meta-model, the operational semantics is defined in a Kermeta resource, and finally the static semantics is brought in an OCL file. Consequently, as illustrated in Figure 9, a designer can create a meta-model defined with the Ecore formalism of the Eclipse Modeling Framework. He can define the static semantics with OCL constraints. Finally with Kermeta, he can define the operational semantics as well as some useful derived features of the meta-models that are called in the OCL specifications. The weaving of all those model fragments is performed automatically in Kermeta, using the *require* instruction as a concrete syntax for this static introduction. Consequently, in the context of the class Main, the meta-model contains the data-structure, the static semantics and the operational semantics.

4.2 Concrete Syntax Issues

Meta-Modeling is a natural approach in the field of language engineering for defining abstract syntaxes. Defining concrete and graphical syntaxes with meta-models is still a challenge. Concrete syntaxes are traditionally expressed with

rules, conforming to EBNF-like grammars, which can be processed by compiler compilers to generate parsers. Unfortunately, these generated parsers produce concrete syntax trees, leaving a gap with the abstract syntax defined by meta-models, and further ad hoc hand-coding is required. We have proposed in [15] a new kind of specification for concrete syntaxes, which takes advantage of meta-models to generate fully operational tools (such as parsers or text generators). The principle is to map abstract syntaxes to concrete syntaxes via bidirectional mapping-models with support for both model-to-text, and text-to-model transformations. Other tools emerge for solving this issue of defining the concrete syntax from a mapping with the abstract syntax like the new Textual Modeling Framework[2]. To get usable graphical editors for your domain specific language (DSL), several projects provides a generative approach to create component and runtime infrastructure for building graphical editors as GMF or TopCaseD. We have used Sintaks and TopCaseD to respectively build the Logo concrete syntax, the Logo graphical syntax and their associated editors.

4.3 Evolution Issues

Thanks to the separation of concerns, constraints and behavior aspects are independent and may be designed in separate resources. Then they can be developed and modified separately. The only consideration during their implementation is that they depend on the structure defined in the Ecore meta-model. Modification to this structure can have consequences that have to be considered in the behavior aspects and in the constraints (if a method signature is changed for example). Here, Kermeta's type system is useful as a way of detecting such incompatible changes at compile time.

5 Related Works

There is a long tradition of basing language tools on grammar formalisms, for example higher order attribute grammars [25]. JastAdd [8] is an example of combining this tradition with object-orientation and simple aspect-orientation (static introductions) to get better modularity mecanisms. With a similar support for object-orientation and static introductions, Kermeta can then be seen as a symetric of JastAdd in the DSML world.

Kermeta cannot really be compared to source transformation systems and languages such as DMS [1], Rascal [12], Stratego [2], or TXL [5] that provide powerful general purpose set of capabilities for addressing a wide range of software analysis problems. Kermeta indeed concentrates on one given aspect of DSML design: adding executability to their meta-models in such a way that any other tool can still be used for advanced analysis or transformation purposes. Still, as illustrated in this paper, Kermeta can also be used to program simple, algorithmic and object-oriented transformations for DSML (e.g.; the *local-to-global* transformation.

[2] `http://www.eclipse.org/modeling/tmf/`

In the world of Modeling, Model Integrated Computing (MIC) [23] is probably the most well known environment centered on the development of DSML. The MIC comprises the following steps:

- Design a Domain Specific Modeling Language (DSML): this step allows engineers to create a language that is tailored to the specific needs of the application domain. One has also to create the tools that can interpret instances of this language (i.e. models of the application),
- this language is then used to construct domain models of the application,
- the transformation tool interprets domain models to build executable models for a specific target platform,

This approach is currently supported by a modeling environment including a tool for designing DSMLs (GME) [6] and a model transformation engine based on graph transformations (GREAT). MIC is a standalone environment for Windows of a great power but also of a great complexity. Kermeta brings in a much more lightweight approach, leveraging the rich ecosystem of Eclipse, and providing the user with advanced composition mechanisms based on the notion of aspect to modularly build his DSML environment within Eclipse.

Another approach builds on the same idea: multi-paradigm modeling. It consists in integrating different modeling languages in order to provide an accurate description of complex systems and simulate them. The approach is supported by the ATOM3 graph transformation engine [24].

Microsoft Software Factories [9] propose the factory metaphor in which development can be industrialized by using a well organized chain of software development tools enabling the creation of products by adapting and assembling standardized parts. A software factory may use a combination of DSMLs to model systems at various levels of abstraction and provide transformation between them. A software factory schema is a directed graph where the nodes are representing particular aspects (called viewpoints) of the system to be modeled and edges represent transformations between them. In particular, viewpoints provide the definition of the DSMLs to be used to create model of the viewpoints, development processes supporting model creation, model refactoring patterns, constraints imposed by other viewpoints (that help ensuring consistency between viewpoints) and finally any artifact assisting the developer in the implementation of models based on viewpoints. Transformations between viewpoints are supported mostly in an hybrid or imperative way through templates, recipes and wizards that are integrated as extensions to Visual Studio. Compared to Software Factories, Kermeta provides an integration platform that makes it much easier to develop independantly and later combine the various aspects of a development environment for a given DSML. Further Kermeta follows OMG standards (MOF, OCL, etc.) and is smootly integrated in the Eclipse platform, that provides an alternative open source IDE to Visual Studio and Software Factories.

In the Eclipse environment, several languages have been developed on top of OCL for model navigation and modification. For instance the Epsilon Object

Language (EOL) [13] is a meta-model independent language that can be used both as a standalone generic model management language or as infrastructure on which task-specific languages can be built. The EOL is not object-oriented (in the sense that it does not define classes itself), even if it is able to manage objects of types defined externally in EMF meta-models in the spirit of JavaScript. In contrast to the EOL, Kermeta is an object-oriented (and aspect-oriented) extension to the EMF, providing full static typing accross the languages it integrates: E-Core, OCL and Kermeta.

6 Conclusion

This article presented the Kermeta platform for building Eclipse based, integrated environments for DSML. Based on an aspect oriented paradigm [16,10] Kermeta has been designed to easily extend meta-models with many different concerns, each expressed in its most appropriate language: MOF for abstract syntax, OCL for static semantics, Kermeta itself for dynamic semantics and model transformations [17], Java for simulation GUI, etc.

Technically, since Kermeta is an extension of MOF, a MOF meta-model can be seen as a valid Kermeta program that just declares packages, classes and so on but *does* nothing. Kermeta can then be used to *breath life* into this meta-model, i.e. transform it into a full blown development environment by introducing nicely modularized aspects for handling concerns of static semantics, dynamic semantics, or model transformations, each coming with Eclipse editor support.

Kermeta is already used in many real life projects: more details are available on www.kermeta.org.

References

1. Baxter, I.D., Pidgeon, C., Mehlich, M.: Dms. In: ICSE, pp. 625–634 (2004)
2. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/xt 0.17. a language and toolset for program transformation. Sci. Comput. Program. 72(1-2), 52–70 (2008)
3. Chauvel, F., Jézéquel, J.-M.: Code generation from UML models with semantic variation points. In: Kent, S., Briand, L. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 54–68. Springer, Heidelberg (2005)
4. Clifton, C., Leavens, G.T.: Multijava: Modular open classes and symmetric multiple dispatch for java. In: OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 130–145 (2000)
5. Cordy, J.R.: The txl source transformation language. Sci. Comput. Program. 61(3), 190–210 (2006)
6. Davis, J.: Gme: the generic modeling environment. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, pp. 82–83. ACM Press, New York (2003)
7. Drey, Z., Faucher, C., Fleurey, F., Vojtisek, D.: Kermeta language reference manual (2006)
8. Ekman, T., Hedin, G.: The jastadd system - modular extensible compiler construction. Sci. Comput. Program. 69(1-3), 14–26 (2007)

9. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, Chichester (2004)
10. Jézéquel, J.-M.: Model driven design and aspect weaving. Journal of Software and Systems Modeling (SoSyM) 7(2), 209–218 (2008)
11. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Liu, Y., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
12. Klint, P., Vinju, J.J., van der Storm, T.: Language design for meta-programming in the software composition domain. In: Bergel, A., Fabry, J. (eds.) SC 2009. LNCS, vol. 5634, pp. 1–4. Springer, Heidelberg (2009)
13. Kolovos, D.S., Paige, R.F., Polack, F.: The epsilon object language (eol). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006)
14. Muller, P.-A.: On Metamodels and Language Engineering. In: From MDD Concepts to Experiments and Illustrations, ISTE (2006) ISBN 1905209592
15. Muller, P.-A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J.-M.: Model-driven analysis and synthesis of concrete syntax. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS/UML 2006. LNCS, vol. 4199, pp. 98–110. Springer, Heidelberg (2006)
16. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving executability into object-oriented meta-languages. In: Kent, S., Briand, L. (eds.) MoDELS/UML 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
17. Muller, P.-A., Fleurey, F., Vojtisek, D., Drey, Z., Pollet, D., Fondement, F., Studer, P., Jézéquel, J.-M.: On executable meta-languages applied to model transformations. In: Model Transformations In Practice Workshop, Montego Bay, Jamaica (October 2005)
18. Object Management Group (OMG). Meta Object Facility (MOF) specification. OMG Document ad/97-08-14 (September1997)
19. Pickin, S., Jard, C., Jéron, T., Jézéquel, J.-M., Le Traon, Y.: Test synthesis from UML models of distributed software. IEEE Transactions on Software Engineering 33(4), 252–268 (2007)
20. Richters, M., Gogolla, M.: OCL: Syntax, semantics, and tools. In: Clark, A., Warmer, J. (eds.) Object Modeling with the OCL. LNCS, vol. 2263, pp. 42–68. Springer, Heidelberg (2002)
21. Saudrais, S., Barais, O., Plouzeau, N.: Integration of time issues into component-based applications. In: Schmidt, H.W., Crnković, I., Heineman, G.T., Stafford, J.A. (eds.) CBSE 2007. LNCS, vol. 4608, pp. 173–188. Springer, Heidelberg (2007)
22. Steel, J., Jézéquel, J.-M.: On model typing. Journal of Software and Systems Modeling (SoSyM) 6(4), 401–414 (2007)
23. Sztipanovits, J., Karsai, G.: Model-integrated computing. IEEE Computer 30(4), 110–111 (1997)
24. Vangheluwe, H., Sun, X., Bodden, E.: Domain-specific modelling with atom3. In: Proceedings of the Second International Conference on Software and Data Technologies, ICSOFT 2007, Barcelona, Spain, July 22-25. vol. PL/DPS/KE/MUSE, pp. 298–304 (2007)
25. Vogt, H., Swierstra, S.D., Kuiper, M.F.: Higher-order attribute grammars. In: PLDI, pp. 131–145 (1989)
26. Ziadi, T., Hëlouët, L., Jézéquel, J.-M.: Towards a UML profile for software product lines. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 129–139. Springer, Heidelberg (2004)

EASY Meta-programming with Rascal

Paul Klint, Tijs van der Storm, and Jurgen Vinju
Centrum Wiskunde & Informatica
and
Universiteit van Amsterdam

Abstract. Rascal is a new language for meta-programming and is intended to solve problems in the domain of source code analysis and transformation. In this article we give a high-level overview of the language and illustrate its use by many examples. Rascal is a work in progress both regarding implementation and documentation. More information is available at http://www.rascal-mpl.org/.

Acknowledgement. This project was carried out as part of ATEAMS, a joint research project team funded by CWI (Netherlands) and INRIA (France).

Keywords: source code analysis, source code transformation, meta-programming, domain-specific languages.

1 A New Language for Meta-programming

Meta-programs are programs that analyze, transform or generate other programs. Ordinary programs work on data; meta-programs work on programs. The range of programs to which meta-programming can be applied is large: from programs in standard languages like C and Java to domain-specific languages for describing high-level system models or applications in specialized areas like gaming or finance. In some cases, even test results or performance data are used as input for meta-programs. Rascal is a new language for *meta-programming*, this is the activity of writing meta-programs.

1.1 The EASY Paradigm

Many meta-programming problems follow a fixed pattern. Starting with some input system (a black box that we usually call *system-of-interest*), first relevant information is extracted from it and stored in an internal representation. This internal representation is then analyzed and used to synthesize results. If the synthesis indicates this, these steps can be repeated over and over again. These steps are shown in Figure 1.1, “EASY: the Extract-Analyze-Synthesize Paradigm”.

This is an abstract view on solving meta-programming problems, but it has wide applicability. Let's illustrate it with a few examples.

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 222–289, 2011.

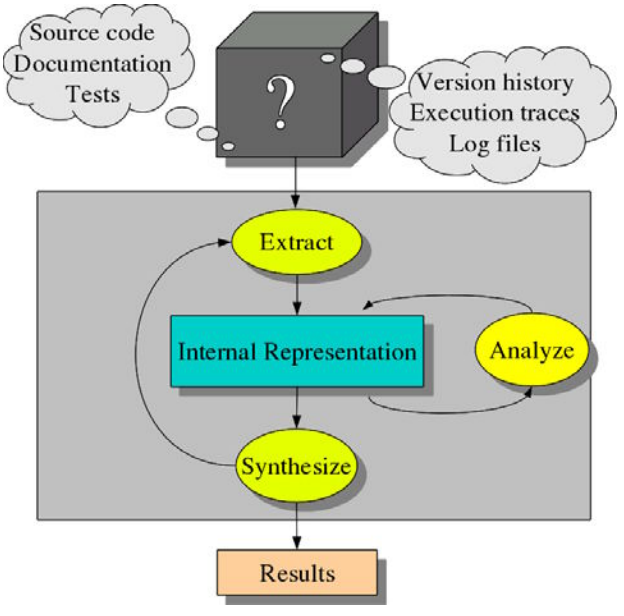


Fig. 1.1. EASY: The Extract-Analyze-Synthesize Paradigm

1.1.1 Finding Security Breaches

Alice is system administrator of a large online marketplace and she is looking for security breaches in her system. The objects-of-interest are the system's log files. First relevant entries are extracted. This will include, for instance, messages from the SecureShell demon that reports failed login attempts. From each entry, login name and originating IP address are extracted and put in a table (the internal representation in this example). These data are analyzed by detecting duplicates and counting frequencies. Finally results are synthesized by listing the most frequently used login names and IP addresses.

1.1.2 A Forensic DSL Compiler

Bernd is a senior software engineer working at a forensic investigation lab of the German government. His daily work is to find common patterns in files stored on digital media that have been confiscated during criminal investigations. Text, audio and video files are stored in many different data formats and each data format requires its own analysis technique. For each new investigation, ad hoc combinations of tools are used. This makes the process very labour-intensive and error-prone. Bernd convinces his manager that designing a new domain-specific language (DSL) for forensic investigations may relieve the pressure on their lab. The DSL should at least be able to define multiple data formats and generate recognizers for them. After designing the DSL---let's call it DERRICK---he makes an EASY implementation for it. Given a DERRICK program for a specific case under investigation, the DERRICK compiler first extracts information from the program and analyzes it further: which media formats

are relevant? Which patterns to look for? How should search results be combined? Given this information, Java code is synthesized that corresponds to the meaning of the DERRICK program: specialized generated code that glues together various existing libraries and tools.

1.1.3 Renovating Financial Software

Charlotte is software engineer at a large financial institution and she is looking for options to connect an old and dusty software system to a web interface. She will need to analyze the sources of that system to understand how it can be changed to meet the new requirements. The objects-of-interest are in this case the source files, documentation, test scripts and any other available information. They have to be parsed in some way in order to extract relevant information, say the calls between various parts of the system. The call information can be represented as a binary relation between caller and callee (the internal representation in this example). This relation with 1-step calls is analyzed and further extended with 2-step calls, 3-step calls and so on. In this way call chains of arbitrary length become available. With this new information, we can synthesize results by determining the entry points of the software system, i.e., the points where calls from the outside world enter the system. Having completed this first cycle, Charlotte may be interested in which procedures can be called from the entry points and so on and so forth. Results will be typically represented as pictures that display the relationships that were found. In the case of source code analysis, a variation of our workflow scheme is quite common. It is then called the extract-analyze-view paradigm and is shown in Figure 1.2, “The extract-analyze-view paradigm”.

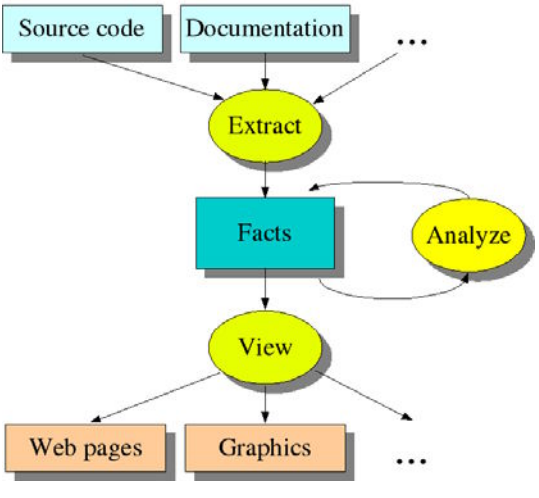


Fig. 1.2. The extract-analyze-view paradigm

1.1.4 Finding Concurrency Errors

Daniel is concurrency researcher at one of the largest hardware manufacturers worldwide. Concurrency is the big issue for his company: it is becoming harder and

harder to make CPUs faster, therefore more and more of them are bundled on a single chip. Programming these multi-core chips is difficult and many programs that worked fine on a single CPU contain hard to detect concurrency errors due to subtle differences in the order of execution that results from executing the code on more than one CPU. He is working on tools for finding concurrency errors. First he extracts facts from the code that are relevant for concurrency problems and have to do with calls, threads, shared variables and locks. Next, he analyzes these facts and synthesizes an abstract model that captures the essentials of the concurrency behaviour of the program. Finally he runs a third-party verification tool with this model as input to do the actual verification.

1.1.5 Model-Driven Engineering

Elisabeth is a software architect at a large airplane manufacturer and her concern is reliability and dependability of airplane control software. She and her team have designed a UML model of the control software and have extended it with annotations that describe the reliability of individual components. She will use this annotated model in two ways: (a) to extract relevant information from it to synthesize input for a statistical tool that will compute overall system reliability from the reliability of individual components; (b) to generate executable code that takes the reliability issues into account.

1.2 Rascal

With these examples in mind, you have a pretty good picture how EASY applies in different use cases. All these cases involve a form of *meta-programming*: software programs (in a wide sense) are the objects-of-interest that are being analyzed, transformed or generated. The Rascal language you are about to learn is designed for meta-programming following the EASY paradigm. It can be applied in domains ranging from compiler construction and implementing domain-specific languages to constraint solving and software renovation.

Since representation of information is central to the approach, Rascal provides a rich set of built-in data types. To support extraction and analysis, parsing and advanced pattern matching are provided. High-level control structures make analysis and synthesis of complex data structures simple.

1.3 Benefits of Rascal

Before you spend your time on studying the Rascal language it may help to first hear our elevator pitch about the main benefits offered by the language:

- **Familiar syntax** in a *what-you-see is-what-you-get style* is used even for sophisticated concepts and this makes the language easy to learn and easy to use.

- **Sophisticated built-in data types** provide standard solutions for many meta-programming problems.

- **Safety** is achieved by finding most errors before the program is executed and by making common errors like missing initializations or invalid pointers impossible. *At the time of writing, this checking is done during execution but a static typechecker is nearing completion.*
- **Local type inference** makes local variable declarations redundant.
- **Pattern matching** can be used to analyze all complex data structures.
- **Syntax definitions** make it possible to define new and existing languages and to write tools for them.
- **Visiting** makes it easy to traverse data structures and to extract information from them or to synthesize results.
- **Templates** enable easy code generation.
- **Functions as values** permit programming styles with high re-use.
- **Generic types** allow writing functions that are applicable for many different types.
- **Eclipse integration** makes Rascal available in a familiar environment: all familiar tools are at your fingertips.

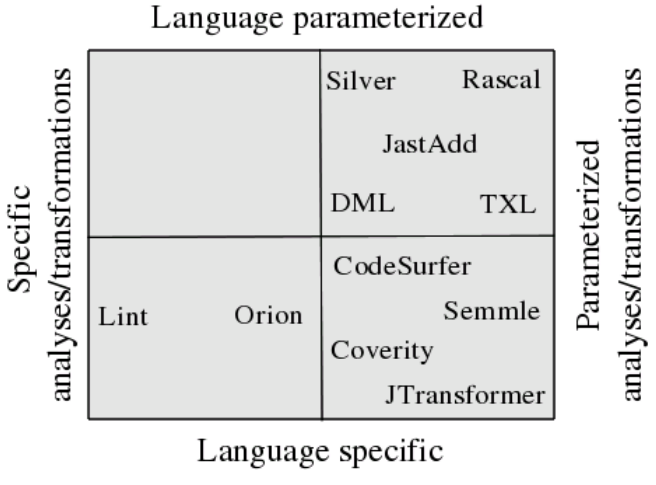


Fig. 1.3. Positioning of various systems

1.4 Related Work

We discuss related tools and specific influences of earlier work on the Rascal language. Figure 1.3, "Positioning of various systems" shows the relative positioning of various systems to be discussed in this section. On the vertical axis, the level of language-

specificity is shown. Systems at the bottom of the figure are language-specific and can only be applied to programs in one specific language. Systems at the top of the figure are language-parameterized, i.e., they are generic and can be applied to arbitrary languages. On the horizontal axis, the level of parameterization of analysis and transformation is displayed. Systems on the left have fixed analysis/transformation properties while the systems more to the right have more and more support for defining analysis and transformation operations.

1.4.1 Related Tool Categories

We focus on positioning Rascal relative to a number of broad categories of tools that are available and give some examples of typical tools in those categories.

Lexical Tools. Lexical tools are based on the use of regular expressions and are supported by widely available systems like Grep, AWK, Perl, Python and Ruby. Regular expressions have limited expressiveness since they cannot express nested structures. Lexical tools are therefore mostly used to extract unstructured textual information, like counting lines of code, collecting statistics about the use of identifiers and keywords, and spotting simple code patterns. Experience shows that maintaining larger collections of regular expressions is hard.

Compiler Tools. Well-known tools like Yacc, Bison, CUP, and ANTLR [Par07] fall in this category. They all use a context-free grammar and generate a parser for the language defined by that grammar. Some provide automation for syntax tree construction but all other operations on parsed programs have to be programmed manually in a standard programming language like C or Java.

Attribute Grammar Tools. An attribute grammar is a context-free grammar enriched with attributes and attribute equations that express dependencies between attributes. In this way semantic operations like typechecking and code generation can be expressed at a high level. Tools in this category are FNC-2 [JPJ+90], JastAdd [HM03], Silver [VWBGK10], and others. Attribute Grammar tools are strong in program analysis but weaker in program transformation.

Relational Analysis Tools. Relational algebra tools provide high-level operations on relational data. They are strong in expressing analysis problems like, for instance, reachability. Examples of tools in this category are GROK [Hol08], Crocopat [Bey06] and RScript [Kli08]. These tools assume that the data are already in relational form and provide support for neither extraction nor transformation.

Transformation Tools. Tools in this category focus on transforming programs from one representation to another representation and they are frequently based on term rewriting. Examples are ASF+SDF [Kli93], [BDH+01], ELAN [BKK+98], STRATEGO [BKVV08], TOM [BBK+07], and TXL [Cor06].

Logic-based Tools. Logic-based languages/tools form a category on their own since they partially overlap with the other categories. The primary example is Prolog that

has been used for syntax analysis, analysis and synthesis and is still in use today. [KHR07] gives an overview of various logic-based approaches. Nonetheless, these tools have never come into wide use. [LR01] illustrates the use of Prolog for analysis and transformation. [KHR07] describes JTransformer, a more recent Prolog-based transformation tool for Java.

Miscellaneous Tools. There is a wide range of tools available in the space sketched in Figure 1.3, "Positioning of various systems". Lint [Joh79] is the classical example of a command-line tool that performs C-specific analysis. ORION [DN04] is another example of a tool for fixed error analysis for C/C++ programs. Codesurfer [AZ05] and Coverity [BBC+10] are examples of tools for the analysis of C programs that provide programmable analyses. Semmle [dMSV+08] is an example of a Java-specific tools with highly-customizable queries. The same holds for JTransformer [KHR07], but this also supports customizable transformations.

Comparing Tool Categories. We audaciously list the strengths and weaknesses of these tool categories in Table 1.1, "Comparing Tool Categories". There are large variations among specific tools in each category but we believe that this table gives at least an impression of the relative strengths of the various categories and also documents the ambition we have with the Rascal language. It is clear from this overview that the level of integration of extraction, analysis and synthesis is mostly weak. Exceptions to this are systems that explicitly integrate these phases. For language-specific systems this are, for example, CodeSurfer [AZ05], Coverity [BBC+10], Semmle and JTransformer. For language-parametric systems examples are DMS, TXL, JastAdd and Silver. All these systems differ in the level of integration and IDE support. Rascal is comparable to the latter systems regarding language-parametricity, provides good IDE support and outperforms many other systems with respect to the integrated and advanced mechanisms for specifying analysis and transformation.

Table 1.1. Comparing Tool Categories

Type	Extract	Analyze	Synthesize
Lexical Tools	++	+/−	−−
Compiler Tools	++	+/−	+/−
Attribute Grammar Tools	++	+/−	−−
Relational Tools	−−	++	−−
Transformation Tools	−−	+/−	++
Logic-based Tools	+/−	+	+
Our Goal: Rascal	++	++	++

1.4.2 Work Directly Related to Rascal

Rascal owes a lot to other meta-programming languages, in particular the user-defined, modular, syntax and term rewriting of ASF+SDF [Kli93], [BDH+01], the relational

calculus as used in RScript [Kli08] and pioneered by GROK [Hol08], traversal functions as introduced in [BKV03], strategies as introduced in ELAN [BKK+98] and Stratego [BKVV08], and integration of term rewriting in Java as done in TOM [BBK+07].

We also acknowledge less specific influences by systems like TXL [Cor06], ANTLR [Par07], JastAdd [HM03], Semmle [dMSV+08], DMS [BPM04], and various others like, for instance, ML and Ruby.

The application of Rascal for refactoring is described in [KvdSV09]. Using Rascal for collecting source code metrics for DSL implementations is presented in [KvdSV10].

1.4.3 Contribution

We claim as main contribution of Rascal the seamless integration of extraction, analysis and synthesis concepts into a relatively simple, statically typed, language with familiar syntax. Specific contributions are the integration of syntax definitions and parsing functions, access to syntax trees both as fully typed and as untyped data structures, the rich collection of data types and patterns, string templates, rewrite rules as normalization device for structured data, and the use of transactions to undo side-effects. All these features are discussed in some detail in this paper.

1.5 Reading Guide

The aim of this article is to give an easy to understand but comprehensive overview of the Rascal language and to offer problem solving strategies to handle realistic problems that require meta-programming. Problems may range from security analysis and model extraction to software renovation, domain-specific language development and code generation.

The scope of this article is limited to the Rascal language and its applications but does not address implementation aspects of the language.

The structure of the description of Rascal is shown in Figure 1.4, “Structure of the Rascal Description”. This article provides a self-contained version of the first three parts:

- **Introduction**: gives a high-level overview of Rascal and consists of Section 1, “A New Language for Meta-Programming” and Section 2, “Rascal Concepts” . It also presents some simple examples in Section 3, “Some Simple Examples”.
- **Problem Solving**: describes the major problem solving strategies in Rascal's application domain, see Section 4, “Problem Solving Strategies”.
- **Examples**: gives a collection of larger examples, see Section 5, “Larger Examples”.

The other parts can be found online at http://www.rascal-mpl.org:

- **Reference**: gives a detailed description of the Rascal language, and all built-in operators and library functions.

- **Support**: gives tables with operators and library functions, a bibliography and a glossary that explains many concepts that are used in the descriptions of Rascal and tries to make them self-contained.

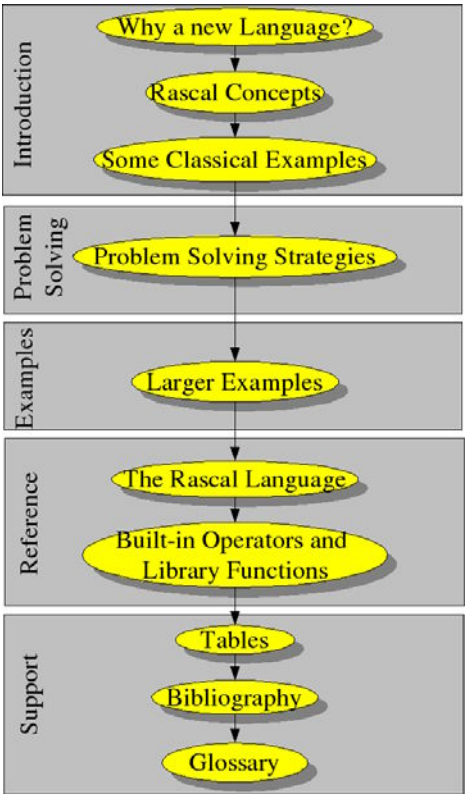


Fig. 1.4. Structure of the Rascal Description

1.6 Typographic Conventions

Rascal code fragments are always shown as a listing like this:

```
  .. here is some Rascal code ...
```

Interactive sessions are shown as a screen like this:

```
rascal> Command;
Type: Value
```

where:

- `rascal>` is the prompt of the Rascal system.
- **`Command`** is an arbitrary Rascal statement or declaration typed in by the user.

- `Type: Value` is the type of the answer followed by the value of the answer as computed by Rascal. In some cases, the response will simply be `ok` when there is no other meaningful answer to give.

2 Rascal Concepts

Before explaining the Rascal language in more detail, we detail our elevator pitch a bit and give you a general understanding of the concepts on which the language is based.

2.1 Values

Values are the basic building blocks of a language and the type of a value determines how they may be used.

Rascal is a value-oriented language. This means that values are immutable and are always freshly constructed from existing parts; sharing and aliasing problems are completely avoided. The language also provides variables. A value can be associated with a variable as the result of an explicit assignment statement: during the lifetime of a variable different (immutable) values may be assigned to it. Other ways to associate a value with a variable are by way of function calls (binding of formal parameters to actual values) and as the result of a successful pattern match.

2.2 Data Types

Rascal provides a rich set of data types:

- Booleans (bool).
- Arbitrary precision integers (int), reals (real) and numbers (num).
- Strings (str) that can act as templates with embedded expressions and statements.
- Source code locations (loc) based on an extension of Universal Resource Identifiers (URI) that allow precise description of text areas in local and remote files.
- Lists (list).
- Tuples (tuple).
- Sets (set).
- Maps (map).
- Relations (rel).
- Untyped tree structures (node)
- Fully typed data structures (data).

The elements of tuples, maps and relation may optionally be labelled for later reference and retrieval. There is a wealth of built-in operators and library functions available on the standard data types.

Table 1.2. Basic Rascal Types

Type	**Examples**
`bool`	`true, false`
`int`	`1, 0, -1, 123456789`
`real`	`1.0, 1.0232e20, -25.5`
`str`	`"abc", "first\nnext", "result: <X>"`
`loc`	`|file:///etc/passwd|`
`datetime`	`$2010-07-15T09:15:23.123+03:00`
`tuple[`T_1`,...,`T_n`]`	`<1,2>, <"john", 43, true>`
`list[`T`]`	`[], [1], [1,2,3], [true, 2, "abc"]`
`set[`T`]`	`{}, {1,2,3,5,7}, {"john", 4.0}`
`rel[`T_1`,...,`T_n`]`	`{<1,2>,<2,3>,<1,3>}, {<1,10,100>, <2,20,200>}`
`map[`T`, `U`]`	`(), (1:true, 2:false), ("a":1, "b":2)`
`node`	`f(), add(x,y), g("abc", [2,3,4])`

The basic Rascal data types are illustrated in Table 1.2, “Basic Rascal Types”. Some types have another type as argument, for instance, `list[int]` denotes a list of integers. In the table, T, T_i and U denote such type arguments.

These built-in data types are closely related to each other:

- In a list all elements have the same static type and the order of elements matters. A list may contain the same value more than once.
- In a set all elements have the same static type and the order of elements does not matter. A set contains an element only once. In other words, duplicate elements are eliminated and no matter how many times an element is added to a set, it will occur in it only once.
- In a tuple all elements (may) have a different static type. Each element of a tuple may have a label that can be used to select that element of the tuple.
- A relation is a set of tuples that all have the same static tuple type.
- A map is an associative table of (key, value) pairs. Key and value (may) have different static types and a key can only be associated with a value once

Untyped trees can be constructed with the built-in type **node**. User-defined algebraic data types (ADTs) allow the introduction of problem-specific types and are a subtype of **node**. A fragment of the abstract syntax for statements (assignment, if, while) in a programming language would look as follows:

```
data STAT = asgStat(Id name, EXP exp)
          | ifStat(EXP exp,list[STAT] thenpart,
                           list[STAT] elsepart)
          | whileStat(EXP exp, list[STAT] body)
          ;
```

Syntax trees that are the result of parsing source files are represented as a data type (Tree). Since all datatypes are a subtype of node, this allows the handling of parse trees both as fully typed ADTs and as untyped nodes (thus enabling generic operations on arbitrary parse trees).

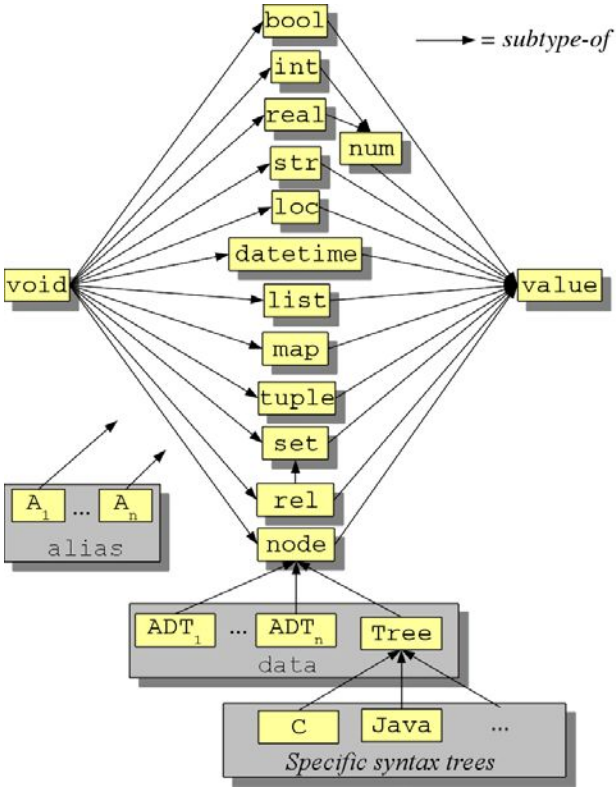


Fig. 1.5. Lattice of Rascal types

Without going into more details, we show the lattice of types in Figure 1.5, “Lattice of Rascal types”: all types are included between the minimal type (void) and the maximal type (value). The relation between node, abstract data type (data), Tree and parse trees

for different language is clearly shown. The alias mechanism allows shorthands for already defined types and is not further discussed.

2.3 Pattern Matching

Pattern matching determines whether a given pattern matches a given value. The outcome can be false (no match) or true (a match). A pattern match that succeeds may bind values to variables.

Pattern matching is *the* mechanism for case distinction (switch statement) and search (visit statement) in Rascal. Patterns can also be used in an explicit match operator `:=` and can then be part of larger boolean expressions. Since a pattern match may have more than one solution, local backtracking over the alternatives of a match is provided. Patterns can also be used in enumerators and control structures like for and while statement.

A very rich pattern language is provided that includes string matching based on regular expressions, matching of abstract patterns, and matching of concrete syntax patterns. Some of the features that are provided are list (associative) matching, set (associative, commutative, idempotent) matching, and deep matching of descendant patterns. All these forms of matching can be used in a single pattern and can be nested. Patterns may contain variables that are bound when the match is successful. Anonymous (don't care) positions are indicated by the underscore (_).

Here is a *regular expression* that matches a line of text, finds the first alphanumeric word in it, and extracts the word itself as well as the text before and after it (`\W` matches all non-word characters; `\w` matches all word characters):

```
/^<before:\W*><word:\w+><after:.*$>/
```

Regular expressions follow the Java regular expression syntax with one exception: instead of using numbered groups to refer to parts of the subject string that have been matched by a part of the regular expression we use the notation:

```
<Name:RegularExpression>
```

If *`RegularExpression`* matches, the matched substring is assigned to string variable *`Name`*.

The following *abstract pattern* matches the abstract syntax of a while statement defined earlier:

```
whileStat(EXP Exp, list[STAT] Stats)
```

Variables in a pattern are either explicitly declared in the pattern itself---as done in the example, e.g., `EXP Exp` declares the variable `Exp` and `list[STAT] Stats` declares the variable `Stats`---or they may be declared in the context in which the pattern occurs. So-called *multi-variables* in list and set patterns are declared by a `*` suffix: `X*` is thus

an abbreviation for `list[...] X` or `set[...] X`, where the precise element type depends on the context. The above pattern can then be written as

```
whileStat(EXP Exp, Stats*)
```

where `Stats*` is an abbreviation for the declaration `list[STAT] Stats`. The choice whether `Stats` is a list or a set depends on the context. If you are not interested in the actual value of the statements the pattern can be written as

```
whileStat(EXP Exp, _*)
```

When there is a grammar for this example language (in the form of an imported SDF definition), we can also write *concrete patterns* as we will see below.

2.4 Enumerators

Enumerators enumerate the values in a given (finite) domain, be it the elements in a list, the substrings of a string, or all the nodes in a tree. Each value that is enumerated is first matched against a pattern before it can possibly contribute to the result of the enumerator. Examples are:

```
int x <- { 1, 3, 5, 7, 11 }
int x <- [ 1 .. 10 ]
/asgStat(Id name, _) <- P
```

The first two produce the integer elements of a set of integers, respectively, a range of integers. Observe that the left-hand side of an enumerator is a pattern, of which `int x` is a specific instance. The use of more general patterns is illustrated by the third enumerator that does a deep traversal (as denoted by the descendant operator `/`) of the complete program `P` (that is assumed to have a `PROGRAM` as value) and only yields statements that match the assignment pattern (`asgStat`). We have defined the `asgStat` constructor earlier in the example in Section 2.2, “Data Types” and here we see how it can be used as pattern.The descendant operator is part of the pattern and has as effect that the pattern is not only tried at the root of the subject to which it is applied (in this case `P`) but also to all its descendants, e.g., subtrees, list elements, and the like. Note the use of an anonymous variable at the `EXP` position in the pattern.

2.5 Comprehensions

Comprehensions are a notation inspired by mathematical set-builder notation that helps to write succinct definitions of lists and sets. They are also inspired by queries as found in a language like SQL.

Rascal generalizes comprehensions in various ways. Comprehensions exist for lists, sets and maps. A comprehension consists of an expression that determines the successive elements to be included in the result and a list of enumerators and tests (boolean expressions). The enumerators produce values and the tests filter them. A standard example is

```
{ x * x | int x <- [1 .. 10], x % 3 == 0 }
```

which returns the set `{9, 36, 81}`, i.e., the squares of the integers in the range `[ 1 .. 10 ]` that are divisible by 3. A more intriguing example is

```
{name | /asgStat(Id name, _) <- P}
```

which traverses program `P` and constructs a set of all identifiers that occur on the left hand side of assignment statements in `P`.

2.6 Control Structures

Control structures like `if` and `while` statements are driven by Boolean expressions, for instance

```
if(N <= 0)
     return 1;
  else
     return N * fac(N - 1);
```

Actually, combinations of generators and Boolean expressions can be used to drive the control structures. For instance,

```
for(/asgStat(Id name, _) <- P, size(name) > 10){
    println(name);
}
```

prints all identifiers in assignment statements (`asgStat`) that consist of more than 10 characters.

2.7 Case Distinction

The switch statement as known from C and Java is generalized: the subject value to switch on may be an arbitrary value and the cases are arbitrary patterns followed by a statement. Here is an example where we take a program P and distinguish two cases for while and if statement:

```
switch (P){
case whileStat(_, _):
     println("A while statement");
case ifStat(_, _, _):
     println("An if statement");
}
```

The switch statement (and related statements like visit) are even more general than their counterpart in languages like C and Java. Each case is comparable to a *transaction*: when the pattern succeeds and the following statement is executed successfully, all changes to variables made by the statement are *committed* and thus become permanent.

The variables bound by the pattern are always local to the statement associated with the case. When a match fails or when the associated statement fails, a *rollback* to the execution point just before the case takes place and all side-effects are undone. External side-effects like I/O and side-effects in user-defined Java code are not undone. Our motivation for this rather heavy mechanism is that it allows a mostly functional programming style even in cases where the statements associated with a matching pattern lead to failure. In those cases, the side-effects are undone and the next pattern of the switch is tried in exactly the same state as the previous one.

2.8 Visiting

Visiting the elements of a data structure is one of the most common operations in our domain and the visitor design pattern is a solution known to every software engineer. Given a tree-like data structure we want to perform an operation on some (or all) nodes of the tree. The purpose of the visitor design pattern is to decouple the logistics of visiting each node from the actual operation on each node. In Rascal the logistics of visiting is completely automated.

Visiting is achieved by way of visit expressions that resemble the switch statement. A visit expression traverses an arbitrarily complex subject value and applies a number of cases (defined in the same way as cases in a switch statement) to all its subtrees. All the elements of the subject are visited and when one of the cases matches the statements associated with that case are executed. These cases may:

- cause some side effect, i.e., assign a value to local or global variables;
- execute an `insert` statement that replaces the current element;
- execute a `fail` statement that causes the match for the current case to fail (and undoing all side-effects due to the successful match itself and the execution of the statements so far).

The value of a visit expression is the original subject value with all replacements made as dictated by matching cases. The traversal order in a visit expressions can be explicitly defined by the programmer. An example of visiting is given in the next subsection and in Section 3.1, “Colored Trees”.

2.9 Functions

Functions allow the definition of frequently used operations. They have a name and formal parameters. They are explicitly declared and are fully typed. Here is an example of a function that counts the number of assignment statements in a program:

```
int countAssignments(PROGRAM P){
    int n = 0;
    visit (P){
    case asgStat(_, _):
         n += 1;
```

```
    }
    return n;
}
```

Functions can also be used as values, thus enabling higher-order functions. Consider the following declarations:

```
int double(int x) { return 2 * x; }

int triple(int x) { return 3 * x; }

int f(int x, int (int) multi){ return multi(x); }
```

The functions double and triple simply multiply their argument with a constant. Function f is, however, more interesting. It takes an integer `x` and a function `multi` (with integer argument and integer result) as argument and applies `multi` to its own argument. `f(5, triple)` will hence return `15`. Function values can also be created anonymously as illustrated by the following, alternative, manner of writing this same call to f:

```
f(5, int (int y){return 3 * y;});
```

Here the second argument of f is an anonymous function.

Rascal is a higher-order language in which functions are first-class values. Our motivation to include first-class functions is to provide sufficient mechanisms for writing re-usable analysis and transformation functions.

2.10 Syntax Definition and Parsing

All source code analysis projects need to extract information directly from the source code. There are two main approaches to this:

- *Lexical information*: Use regular expressions to extract useful, but somewhat superficial, flat, information.
- *Structured information*: Use syntax analysis to extract the complete, nested, structure of the source code in the form of a syntax tree.

In Rascal, we reuse the Syntax Definition Formalism (SDF) and its tooling. See http://www.rascal-mpl.org/ [http://www.meta-environment.org/Meta-Environment/Documentation] for tutorials and manuals for SDF.

SDF modules define grammars and these modules can be imported in a Rascal module. These grammar rules can be applied in writing concrete patterns to match parts of parsed source code. Here is an example of the same while-pattern we saw in Section 2.3, “Pattern Matching”, but now in concrete form:

```
while <Exp> do <Stats> od
```

Importing an SDF module has the following effects:

- All non-terminals (*sorts* in SDF jargon) that are used in the imported grammar are implicitly declared as Rascal types. For each SDF sort `S`, composite symbols like `S*` and `{S ","}+` also become available as types. This makes it possible to handle parse trees and parse tree fragments as fully typed values. They can be assigned to variables, can be stored in larger data structures, can be passed as arguments to functions and can be used in pattern matching.
- For all *start symbols* of the grammar, *parse functions* are implicitly declared that can parse source files according to a specific start symbol.
- Concrete syntax patterns for that specific grammar can be used.
- Concrete syntax constructors can be used that allow the construction of new parse trees.

The following example parses a Java compilation unit from a text file and counts the number of method declarations:

```
module Count
import languages::java::syntax::Java;
import ParseTree;

public int countMethods(loc file){
  int n = 0;
  for(/MethodDeclaration md <- parse(#CompilationUnit,
                                     file))
      n += 1;
  return n;
}
```

First observe that importing the Java grammar has as effect that non-terminals like `MethodDeclaration` and `CompilationUnit` become available as types in the Rascal program.

The implicitly declared function `parse` takes a reified type (`#CompilationUnit`, recall that all non-terminals are implicitly declared as types) and a location as arguments and parses the contents of the location according to the given non-terminal. Next, a match for embedded `MethodDeclarations` is done in the enumerator of the for statement. This example ignores many potential error conditions (like error in opening or reading the file, or syntax error during parsing) but does illustrate some of Rascal's syntax and parsing features.

2.11 Rewrite Rules

A *rewrite rule* is a recipe for simplifying values. An example is `(a + b)^2 = a^2 + 2ab + b^2`. A rewrite rule has a pattern as left-hand side (here: `(a + b)^2`) and a

replacement as right-hand side (here: $a^2 + 2ab + b^2$). Given a value and a set of rewrite rules the patterns are tried on every subpart of the value and replacements are made if a match is successful. This is repeated as long as some pattern matches.

Rewrite rules are the only implicit control mechanism in the language and are used to maintain invariants during computations. For example, in a package for symbolic differentiation it is desirable to keep expressions in simplified form in order to avoid intermediate results like `sum(product(num(1), x), product(num(0), y))` that can be simplified to `x`. The following rules achieve this:

```
rule simplify1 product(num(1), Expression e) => e;
rule simplify2 product(Expression e, num(1)) => e;
rule simplify3 product(num(0), Expression e) => num(0);
rule simplify4 product(Expression e, num(0)) => num(0);
rule simplify5 sum(num(0), Expression e)     => e;
rule simplify6 sum(Expression e, num(0))     => e;
```

Whenever a new value of type `Expression` is constructed, these rules are *implicitly* applied to that expression and all its subexpressions. When a pattern at the left-hand side of a rule applies, the matching subexpression is replaced by the right-hand side of the rule. This is repeated as long as any rule can be applied.

Rewrite rules are activated automatically when a value of some specific type is created and one may always assume that values of that type are in simplified form, i.e., all applicable rewrite rules have been applied to it.

Rewrite rules are *Turing complete*, in other words any computable function can be defined using rewrite rules, including functions that do not terminate. The programmer should be aware of this when defining rewrite rules.

2.12 Equation Solving

Many problems can be solved by forms of *constraint solving*. This is a declarative way of programming: specify the constraints that a problem solution should satisfy and how potential solutions can be generated. The actual solution (if any) is found by enumerating possible solutions and testing their compliance with the constraints.

Rascal provides a `solve` statement that helps writing constraint solvers. A typical example is dataflow analysis where the propagation of values through a program can be described by a set of equations. Their solution can be found with the `solve` statement. See Section 5.6, “Dataflow Analysis” for examples.

2.13 Other Features

All language features (including the ones just mentioned) are described in more detail later on in this article. Some features we have not yet mentioned are:

- A Rascal programs consists of a set of modules that are organized in packages.

- A module can import other modules. These can be Rascal modules or SDF modules (as shown above in Section 2.10, “Syntax Definition and Parsing”).
- The visibility of an entity declared in a module can be controlled using a public/private modifier.
- A data structures may have annotations that can be explicitly used and modified.
- There is an extensive library for built-in data types, input/output, fact extraction from Java source code, visualization, and more.

2.14 Typechecking and Execution

Rascal has a statically checked type system that prevents type errors and uninitialized variables at runtime. There are no runtime type casts as in Java and there are therefore fewer opportunities for run-time errors. The language provides *higher-order, parametric polymorphism*. A type aliasing mechanism allows documenting specific uses of a type. Built-in operators are heavily overloaded. For instance, the operator + is used for addition on integers and reals but also for list concatenation, set union etc.

The flow of Rascal program execution is completely explicit. Boolean expressions determine choices that drive the control structures. Rewrite rules form the only exception to the explicit control flow principle. Only local backtracking is provided in the context of boolean expressions and pattern matching; side effects are undone in case of backtracking.

3 Some Simple Examples

The following simple examples will help you to grasp the main features of Rascal quickly. You can also consult the online documentation at http://www.rascal-mpl.org/ for details of the language or specific operators or functions.

3.1 Colored Trees

Suppose we have binary trees---trees with exactly two children--that have integers as their leaves. Also suppose that our trees can have red and black nodes. Such trees can be defined as follows:

```
module demo::ColoredTrees

data ColoredTree =
            leaf(int N)
          | red(ColoredTree left, ColoredTree right)
          | black(ColoredTree left, ColoredTree right);
```

We can use them as follows:

```
rascal> import demo::ColoredTrees;
```

```
ok

rascal> rb = red(black(leaf(1), red(leaf(2),leaf(3))),
                 black(leaf(3), leaf(4)));
ColoredTree: red(black(leaf(1),red(leaf(2),leaf(3))),
                 black(leaf(3),leaf(4)))
```

Observe that the type of variable `rb` was automatically inferred to be `ColoredTree`.

We define two operations on ColoredTrees, one to count the red nodes, and one to sum the values contained in all leaves:

```
// continuing module demo::ColoredTrees

public int cntRed(ColoredTree t){
   int c = 0;
   visit(t) {
     case red(_,_): c = c + 1;❶
   };
   return c;
}

public int addLeaves(ColoredTree t){
   int c = 0;
   visit(t) {
     case leaf(int N): c = c + N;❷
   };
   return c;
}
```

❶ Visit all the nodes of the tree and increment the counter `c` for each red node.
❷ Visit all nodes of the tree and add the integers in the leaf nodes.

This can be used as follows:

```
rascal> cntRed(rb);
int: 2
rascal> addLeaves(rb);
int: 13
```

A final touch to this example is to introduce green nodes and to replace all red nodes by green ones:

```
// continuing module demo::ColoredTrees

data ColoredTree = green(ColoredTree left,
                         ColoredTree right);❶
```

```
public ColoredTree makeGreen(ColoredTree t){
   return visit(t) {
     case red(l, r) => green(l, r)      ❷
   };
}
```

❶ Extend the ColoredTree data type with a new `green` constructor.
❷ Visit all nodes in the tree and replace red nodes by green ones. Note that the variables `l` and `r` are introduced here without a declaration.

This is used as follows:

```
rascal> makeGreen(rb);
ColoredTree: green(black(leaf(1),green(leaf(2),leaf(3))),
                   black(leaf(3),leaf(4)))
```

This example illustrates the following:

- Abstract data types can be extended as shown by adding the `green` constructor. In large applications this is often encountered as dialects/variants/extensions of a base data type.
- Using `visit` one only has to consider the constructors of interest (e.g., the `leaf` nodes or the `red` nodes) as opposed to all constructors of a datatype. For large data types describing, for instance, programming languages or file formats this can make a large difference in the number of cases that has to be specified.

3.2 Word Replacement

Suppose you are in the publishing business and are responsible for the systematic layout of publications. Authors do not systematically capitalize words in titles---"Word replacement" instead of Word Replacement"--- and you want to correct this. We solve this problem in two steps. The first step is to define the capitalization of a single word:

```
module demo::WordReplacement
import String;

public str capitalize(str word)
{
   if(/^<letter:[a-z]><rest:.*$>/ := word) ❶
      return toUpperCase(letter) + rest;❷
   else
     return word;❸
}
```

❶ The function capitalize takes a string as input and capitalizes its first character if that is a letter. This is done using a regular expression match that anchors the

match at the beginning (^), expects a single letter and assigns it to the variable letter (`letter:[a-z]`) followed by an arbitrary sequence of letters until the end of the string that is assigned to the variable rest (`<rest:.*$>`).

❷ If the regular expression matches we return a new string with the first letter capitalized.

❸ Otherwise we return the word unmodified.

The second step is to capitalize all the words in a string. Here are two solutions:

```
// continuing module demo::WordReplacement

public str capAll1(str S)
{
 result = "";
 while (/^<before:\W*><word:\w+><after:.*$>/ := S) { ❶
    result += before + capitalize(word);
    S = after;
  }
  return result;
}

public str capAll2(str S)
{
   return visit(S){❷
     case /<word:\w+>/i ❸ => capitalize(word)❹
   };
}
```

❶ In the first solution `capAll1` we just loop over all the words in the string and capitalize each word. The variable `result` is used to collect the successive capitalized words. Here we use `\W` do denote non-word characters and `\w` for word characters.

❷ In the second solution we use a visit expression to visit all the substrings of S. Each matching case advances the substring by the length of the pattern it matches and replaces that pattern by another string. If no case matches the next substring is tried.

❸ The single case matches a word (note that `\w` matches a word character).

❹ When the case matches a word, it is replaced by a capitalized version. The modifier `i` at the end of the regular expressions denotes case-insensitive matching.

We can apply this all as follows:

```
rascal> import demo::WordReplacement;
ok

rascal> capitalize("rascal");
```

```
str: "Rascal"

rascal> capAll1("rascal is great");
str: "Rascal Is Great"
```

This example illustrates that regular expressions are an integral part of the language and can be used in control statements (`if`, `while`), explicit pattern match (`:=`) and as patterns in cases of a `visit`. Their use in the EASY domain is:

- Extracting lexical facts from source code (e.g., identifiers, call statements).
- Parsing DSLs with a strong lexical bias, e.g., markup languages.
- Matching lexical entities in a parse tree, e.g., find all assignments to variables containing the substring "`year`".

3.3 Template Programming

Many websites and code generators use template-based code generation. They start from a text template that contains embedded variables and code. The template is "executed" by replacing the embedded variables and code by their string value. A language like PHP is popular for this feature. Let's see how we can do this in Rascal. Given a mapping from field names to their type, the task at hand is to generate a Java class that contains those fields and corresponding getters and setters. Given a mapping

```
public map[str, str] fields = (
   "name" : "String",
   "age" : "Integer",
   "address" : "String"
);
```

we want the call

```
genClass("Person", fields)
```

to produce the following output:

```
    public class Person {

        private Integer age;
        public void setAge(Integer age) {
          this.age = age;
        }
        public Integer getAge() {
          return age;
        }
```

```
        private String name;
        public void setName(String name) {
          this.name = name;
        }
        public String getName() {
          return name;
        }

        private String address;
        public void setAddress(String address) {
          this.address = address;
        }
        public String getAddress() {
          return address;
        }

    }
```

This is achieved by the following definition of `genClass`:

```
module demo::StringTemplate

import String;

public str capitalize(str s) {
  return toUpperCase(substring(s, 0, 1)) +
         substring(s, 1);
}

public str genClass(str name, map[str,str] fields) {
 return "
   public class <name > {
     <for (x <- fields) {
       str t = fields[x];
       str n = capitalize(x);>
       private <t> <x>;
       public void set<n>(<t> <x>) {
         this.<x> = <x>;
       }
       public <t> get<n>() {
         return <x>;
       }
     <}>
   }
";
}
```

This code makes extensive use of Rascal's string interpolation mechanism. All characters inside the string quotes (" and ") are taken literally, except for interpolations that are indicated by angle brackets(< and >). An interpolation is an expression that is evaluated when the string is constructed; the value of the expression replaces the interpolation in the resulting string. Interpolation may contain nested interpolations. The above example contains several examples:

- The interpolation `<name >` consists of a single variable and will, in this example, be replaced by `"Person "`. Observe that spaces and layout are significant inside interpolations.
- The interpolation `<for (x <- fields){ str t = fields[x]; str n = capitalize(x);> ... <}>` consists of a complete for statement that will generate all field names, getters and setters. Observe that the statement consists of three parts:
  - The header <for (x <- fields){ str t = fields[x]; str n = capitalize(x);>. Inside the header spaces are not significant.
  - The body `private <t> <x> ...` . Inside the body spaces are significant. Embedded interpolations may occur inside the body. Observe how these embedded interpolations access the map fields and customize the template for a Java class.
  - The closing brace <}> of the for statement.

This example illustrates how string templates can be used for code generation tasks.

3.4 A Domain-Specific Language for Finite State Machines

Finite State Machines (FSMs) are a universal device in Computer Science and are used to model problems ranging from lexical tokens to concurrent processes. An FSM consists of named states and labeled transitions between states. An example is shown in Figure 1.6, “Example of a Finite State Machine”. This example was suggested by G. Hedin at GTTSE09.

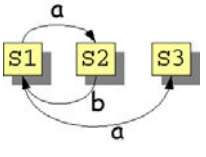


Fig. 1.6. Example of a Finite State Machine

This same information can be represented in textual form as follows:

```
finite-state machine
    state S1;
```

```
    state S2;
    state S3;
    trans a: S1 -> S2;
    trans b: S2 -> S1;
    trans a: S1 -> S3
```

and here is where the idea is born to design a Domain-Specific Language for finite state machines (aptly called FSM). The design of a DSL always proceeds in three steps:

1. **Do domain analysis**. Explore the domain and make an inventory of the relevant concepts and their interactions.

2. **Define syntax**. Design a textual syntax to represent these concepts and interactions.

3. **Define operations**. Define operations on DSL programs. This may be, for example, typechecking, validation, or execution.

We will now apply these steps to the FSM domain.

Do domain analysis. We assume that the FSM domain is sufficiently known. The concepts are states and labeled transitions.

Define syntax. We define a textual syntax for FSMs. This syntax is written in the Syntax Definition Formalism SDF. See http://www.meta-environment.org/Meta-Environment/Documentation for tutorials and manuals for SDF. The syntax definition looks as follows:

```
module demo/StateMachine/Syntax

imports basic/Whitespace
imports basic/IdentifierCon

exports
  context-free start-symbols
    FSM

  sorts FSM Decl Trans State IdCon

  context-free syntax
    "state" IdCon                          -> State
    "trans" IdCon ":" IdCon  "->" IdCon    -> Trans
    State                                  -> Decl
    Trans                                  -> Decl
    "finite-state" "machine" {Decl ";"}+   -> FSM
```

Two standard modules for whitespace and identifiers are imported and next a fairly standard grammar for state machines is defined. Observe that SDF syntax rules are

written in (horizontally) reverse order as compared to standard BNF notation, with the nonterminal appearing on the right.

Define Operations. There are various operations one could define on a FSM: executing it for given input tokens, reducing a non-deterministic automaton to a deterministic one, and so on. Here we select a reachability check on FSMs as an example.

We start with the usual imports and define a function `getTransitions` that extracts all transitions from an FSM:

```
module demo::StateMachine::CanReach

import demo::StateMachine::Syntax;
import Relation;
import Map;
import IO;

// Extract from a given FSM all transitions as a relation

public rel[str, str] getTransitions(FSM fsm){
   return
   {<"<from>", "<to>"> |
    /`trans <IdCon a>: <IdCon from> -> <IdCon to>` <- fsm
   };
}
```

The function `getTransitions` illustrates several issues. Given a concrete FSM, a deep pattern match (/) is done searching for `trans` constructs. Since FSM is a type that corresponds to a nonterminal, we use a concrete pattern to achieve this: it is enclosed by backquotes (` and `) and consists of a grammar rule with embedded variable declarations. For each match three identifiers (`IdCon`) are extracted and assigned to the variables `a`, `from`, respectively, `to`. Next `from` and `to` are converted to a string (using the string interpolations `"<from>"` and `"<to>"`) and finally they are placed in a tuple in the resulting relation. The net effect is that transitions encoded in the syntax tree of `fsm` are collected in a relation for further processing.

Next, we compute all reachable states in the function `canReach`:

```
// continuing module demo::StateMachine::CanReach

public map[str, set[str]] canReach(FSM fsm){
  transitions = getTransitions(fsm);
  return
    ( s: (transitions+)[s] |
      str s <- carrier(transitions)
    );
}
```

Here `str s <- carrier(transitions)` enumerates all elements that occur in the relations that is extracted from `fsm`. A map comprehension is used to construct a map from each state to all states that can be reached it. Transitive closure is denoted by a postfix +-operator and `transitions+` is thus the transitive closure of the transition relation and `(transitions+)[s]` gives the image of that closure for a given state; in other words all states that can be reached from it.

Finally, we declare an example FSM (observe that it uses FSM syntax in Rascal code!):

```
// continuing module demo::StateMachine::CanReach

public FSM example =
       `finite-state machine
          state S1;
          state S2;
          state S3;
          trans a: S1 -> S2;
          trans b: S2 -> S1;
          trans a: S1 -> S3`;
```

Testing the above functions gives the following results:

```
rascal> import demo::StateMachine::CanReach;
ok

rascal> getTransitions(example);

rel[str,str]: {<"S1", "S2">, <"S2", "S1">, <"S1", "S3">}
rascal> canReach(example);

map[str: set[str]: ("S1" : {"S1", "S2", "S3"},
                    "S2" : {"S1", "S2", "S3"},
                    "S3" : {})
```

This example illustrates:

- The use of concrete syntax to define a DSL, e.g., `demo/StateMachine/Syntax`.
- The use of non-terminals of the grammar as types in a Rascal program, e.g., `FSM`.
- The use of quoted DSL fragments in Rascal code, e.g., `example`.
- Extraction of information from a parsed DSL program, e.g., `getTransitions`.
- Representation of this information as relation.
- Description of a reachability check on this relation (e.g., `canReach`).

4 Problem Solving Strategies

Before we study more complicated examples, it is useful to discuss some general problem solving strategies that are relevant in Rascal's application domain. Rascal and supporting libraries and tools were specifically designed to support these strategies.

To appreciate these general strategies, it is good to keep some specific problem areas in mind:

- **Documentation generation**: extract facts from source code and use them to generate textual documentation. A typical example is generating web-based documentation for legacy languages like Cobol and PL/I.
- **Metrics calculation**: extract facts from source code (and possibly other sources like test runs) and use them to calculate code metrics. Examples are cohesion and coupling of modules and test coverage.
- **Model extraction**: extract facts from source code and use them to construct an abstract model of the source code. An example is extracting lock and unlock calls from source code and building an automaton that guarantees that lock/unlock occurs in pairs along every control flow path.
- **Model-based code generation**: given a high-level model of a software system, described in UML or some other modelling language, transform this model into executable code. UML-to-Java code generation falls in this category.
- **Source-to-source transformation**: given certain objectives like removing deprecated language features, upgrading to newer APIs and the like, perform large-scale, fully automated, source code transformation.
- **Interactive refactoring**: given known "code smells" allow a user to interactively indicate how these smells should be removed. The refactoring features in Eclipse and Visual Studio are examples.

With these examples in mind, we can study the overall problem solving workflow as shown in Figure 1.7, “General 3-Phased Problem Solving Workflow”. It consists of three optional phases:

- If **extraction** is needed to solve the problem, then define the extraction phase, see Section 4.1, “Defining Extraction”.
- If **analysis** is needed, then define the analysis phase, see Section 4.2, “Defining Analysis”.
- If **synthesis** is needed, then define the synthesis phase, see Section 4.3, “Defining Synthesis”.

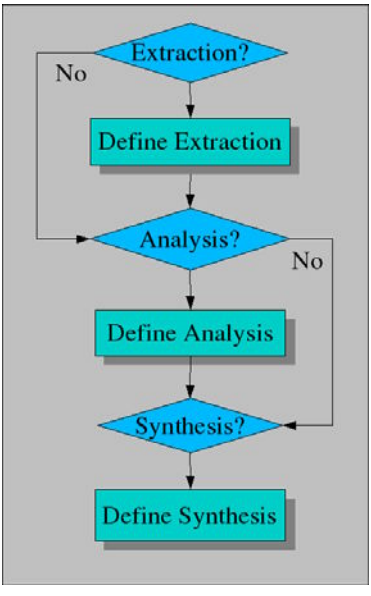


Fig. 1.7. General 3-Phased Problem Solving Workflow

Each phase is subject to a validation and improvement workflow as shown in Figure 1.8, "Validation and Improvement Workflow". Each individual phase as well as the combination of phases may introduce errors and has thus to be carefully validated. In combination with the detailed strategies for each phase, this forms a complete approach for problem solving and validation using Rascal.

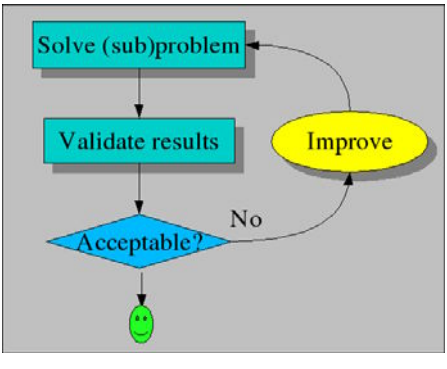


Fig. 1.8. Validation and Improvement Workflow

A major question in every problem solving situation is how to determine the requirements for each phase of the solution. For instance, how do we know what to extract from the source code if we do not know what the desired end results of the project are? The standard solution is to use a workflow for requirements gathering that

is the inverse of the phases needed to solve the complete problem. This is shown in Figure 1.9, “Requirements Workflow” and amounts to the phases:

- **Requirements of the synthesis phase**. This amounts to making an inventory of the desired results of the whole project and may include generated source code, abstract models, or visualizations.

- **Requirements of the analysis phase**. Once these results of the synthesis phase are known, it is possible to list the analysis results that are needed to synthesize desired results. Possible results of the analysis phase include type information, structural information of the original source.

- **Requirements of the extraction phase**. As a last step, one can make an inventory of the facts that have to be extracted to form the starting point for the analysis phase. Typical facts include method calls, inheritance relations, control flow graphs, usage patterns of specific library functions or language constructs.

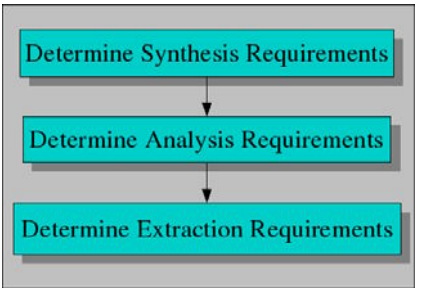


Fig. 1.9. Requirements Workflow

You will have to identify requirements for each phase when you apply them to a specific example from the list given earlier.

When these requirements have been established, it becomes much easier to actually carry out the project using the three phases of Figure 1.7, “General 3-Phased Problem Solving Workflow”.

4.1 Defining Extraction

How can we extract facts from the *System under Investigation* (SUI) that we are interested in? The extraction workflow is shown in Figure 1.10, “Extraction Workflow” and consists of the following steps:

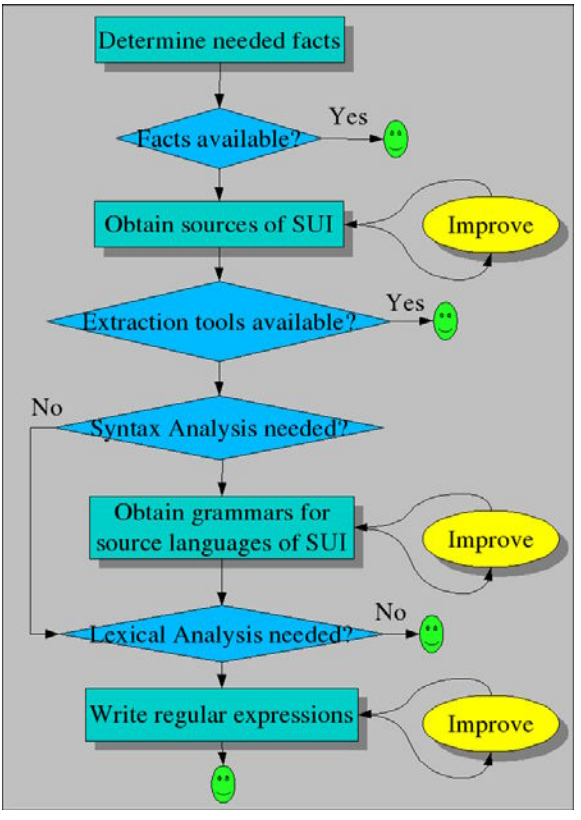


Fig. 1.10. Extraction Workflow

- First and foremost we have to determine which facts we need. This sounds trivial, but it is not. A common approach is to use look-ahead and to sketch the queries that are likely to be used in the analysis phase and to determine which facts are needed for them. Start with extracting these facts and refine the extraction phase when the analysis phase is completely defined.

- If relevant facts are already available (and they are reliable!) then we are done. This may happen when you are working on a system that has already been analyzed by others.

- Otherwise you need the source code of the SUI. This requires:

  - Checking that all sources are available (and can be compiled by the host system on which they are usually compiled and executed). Due to missing or unreliable configuration management on the original system this may be a labour-intensive step that requires many iterations.
  - Determining in which languages the sources are written. In larger systems it is common that three or more different languages are being used.
- If there are reliable third-party extraction tools available for this language mix, then we only have to apply them and we are done. Here again, validation is needed that the extracted facts are as expected.
- The extraction may require syntax analysis. This is the case when more structural properties of the source code are needed such as the flow-of-control, nesting of declarations, and the like. There two approaches here:
  - Use a third-party parser, convert the source code to parse trees and do the further processing of these parse trees in Rascal. The advantage is that the parser can be re-used, the disadvantage is that data conversion is needed to adapt the generated parse tree to Rascal. Validate that the parser indeed accepts the language the SUI is written in, since you will not be the first who has been bitten by the language dialect monster when it turns out that the SUI uses a local variant that slightly deviates from a mainstream language.
  - Use an existing SDF definition of the source language or write your own definition. In both cases you can profit from Rascal's seamless integration with SDF. Be aware, however, that writing a grammar for a non-trivial language is a major undertaking and may require weeks to months of work. Whatever approach you choose, validate the resulting grammar.
- The extraction phase may only require lexical analysis. This happens when more superficial, textual, facts have to be extracted like procedure calls, counts of certain statements and the like. Use Rascal's full regular expression facilities to do the lexical analysis.

It may happen that the facts extracted from the source code are *wrong*. Typical error classes are:

- Extracted facts are *wrong*: the extracted facts incorrectly state that procedure `P` calls procedure `Q` but this is contradicted by a source code inspection. This may happen when the fact extractor uses a conservative approximation when precise information is not statically available. In the C language, when procedure P performs an indirect call via a pointer variable, the approximation may be that P calls all procedures in the program.
- Extracted facts are *incomplete*: for example, the inheritance between certain classes in Java code is missing.

The strategy to validate extracted facts differs per case but here are three possibilities:

- Post process the extracted facts (using Rascal, of course) to obtain trivial facts about the source code such as total lines of source code and number of procedures, classes, interfaces and the like. Next validate these trivial facts with tools like `wc` (word and line count), `grep` (regular expression matching) and others.
- Do a manual fact extraction on a small subset of the code and compare this with the automatically extracted facts.
- Use another tool on the same source and compare results whenever possible. A typical example is a comparison of a call relation extracted with different tools.

The Rascal features that are most frequently used for extraction are:

- Regular expression patterns to extract textual facts from source code.
- Syntax definitions and concrete patterns to match syntactic structures in source code.
- Pattern matching (used in many Rascal statements).
- Visits to traverse syntax trees and to locally extract information.
- The repertoire of built-in data types (like lists, maps, sets and relations) to represent the extracted facts.

A large diversity of problems can be encountered in the extraction phase. Rascal tries to provide a uniform framwork with the right tools to solve them.

4.2 Defining Analysis

The analysis workflow is shown in Figure 1.11, “Analysis Workflow” and consists of two steps:

- Determine the results that are needed for the synthesis phase.
- Write the Rascal code to perform the analysis. This may amount to:
  - Reordering extracted facts to make them more suitable for the synthesis phase.
  - Enriching extracted facts. Examples are computing transitive closures of extracted facts (e.g., A may call B in one or more calls), or performing data reduction by abstracting aways details (i.e., reducing a program to a finite automaton).
  - Combining enriched, extracted, facts to create new facts.

As before, validate, validate and validate the results of analysis. Essentially the same approach can be used as for validating the facts. Manual checking of answers on random

samples of the SUI may be mandatory. It also happens frequently that answers inspire new queries that lead to new answers, and so on.

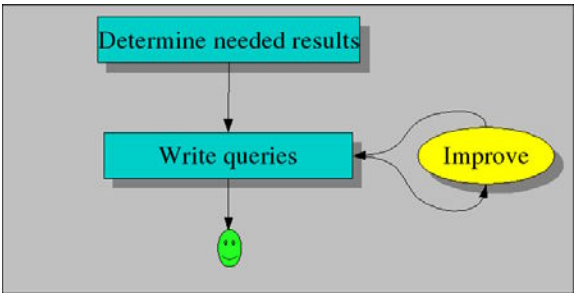


Fig. 1.11. Analysis Workflow

The Rascal features that are frequently used for analysis are:

- List, set and map comprehensions.
- The built-in operators and library functions, in particular for lists, maps, sets and relations.
- Pattern matching (used in many Rascal statements).
- Visits and switches to further process extracted facts.
- The solve statement for constraint solving.
- Rewrite rules to simplify results and to enforce constraints.

4.3 Defining Synthesis

Results are synthesized as shown in Figure 1.12, “Synthesis Workflow”. This consists of the following steps:

- Determine the results of the synthesis phase. A wide range of results is possible including:
  - Generated source code.
  - Generated abstract representations, like finite automata or other formal models that capture properties of the SUI.
  - Generated data for visualizations that will be used by visualization tools.
- If source code is to be generated, there are various options.
  - Print strings that are customized using string interpolation.

- First generate the desired output in the form of an abstract syntax tree and then convert this tree to a string (perhaps using forms of pretty printing).
- Generate the desired output directly as syntactically correct structured text. This is achieved by using a grammar of the target source language and composing syntactically correct program fragments during code generation. The grammar defines the structure of parse trees for the target language and guarantees the construction of type correct parse tree fragments. This approach hence guarantees the generation of syntactically correct source code as opposed to code generation using print statements or string templates.

- If other output is needed (e.g., an automaton or other formal structure) write data declarations to represent that output.
- Finally, write functions and rewrite rules that generate the desired results.

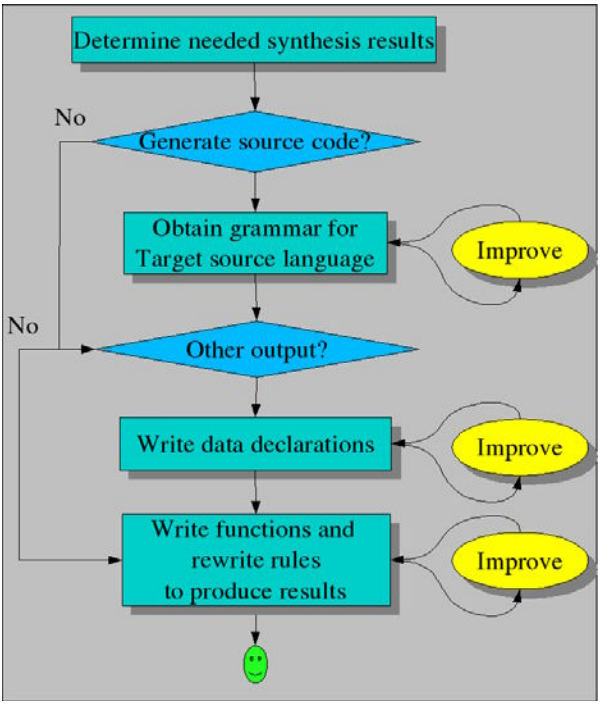


Fig. 1.12. Synthesis Workflow

The Rascal features that are frequently used for synthesis are:

- Syntax definitions or data declarations to define output formats.
- Pattern matching (used in many Rascal statements).

- Visits of data structures and on-the-fly code generation.
- Rewrite rules.

5 Larger Examples

Now we will have a closer look at some larger applications of Rascal. We start with a call graph analysis in Section 5.1, “Call Graph Analysis” and then continue with the analysis of the component structure of an application in Section 5.2, “Analyzing the Component Structure of an Application” and of Java systems in Section 5.3, “Analyzing the Structure of Java Systems”. Next we move on to the detection of uninitialized variables in Section 5.4, “Finding Uninitialized and Unused Variables in a Program”. As an example of computing code metrics, we describe the calculation of McCabe's cyclomatic complexity in Section 5.5, “McCabe Cyclomatic Complexity”. Several examples of dataflow analysis follow in Section 5.6, “Dataflow Analysis”. A description of program slicing concludes the chapter, see Section 5.7, “Program Slicing”.

5.1 Call Graph Analysis

Suppose a mystery box ends up on your desk. When you open it, it contains a huge software system with several questions attached to it:

- How many procedure calls occur in this system?
- How many procedures does it contains?
- What are the entry points for this system, i.e., procedures that call others but are not called themselves?
- What are the leaves of this application, i.e., procedures that are called but do not make any calls themselves?
- Which procedures call each other indirectly?
- Which procedures are called directly or indirectly from each entry point?
- Which procedures are called from all entry points?

Let's see how these questions can be answered using Rascal.

5.1.1 Preparations

To illustrate this process consider the workflow in Figure 1.13, “Workflow for analyzing mystery box”. First we have to extract the calls from the source code. Rascal is very good at this, but to simplify this example we assume that this call graph has already been extracted. Also keep in mind that a real call graph of a real application will contain thousands and thousands of calls. Drawing it in the way we do later on in Figure 1.14, “Graphical representation of the `calls` relation” makes no sense since we get a uniformly black picture due to all the call dependencies. After the extraction phase,

we try to understand the extracted facts by writing queries to explore their properties. For instance, we may want to know *how many calls* there are, or *how many procedures*. We may also want to enrich these facts, for instance, by computing who calls who in more than one step. Finally, we produce a simple textual report giving answers to the questions we are interested in.

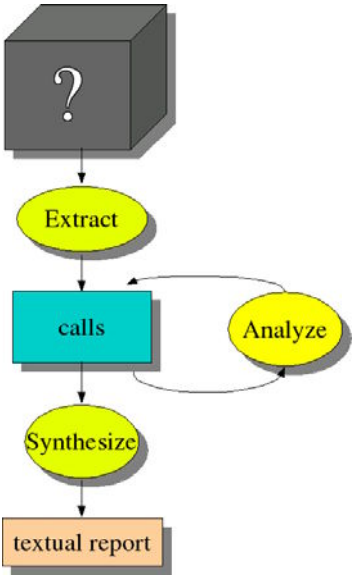


Fig. 1.13. Workflow for analyzing mystery box

Now consider the call graph shown in Figure 1.14, "Graphical representation of the `calls` relation". This section is intended to give you a first impression what can be done with Rascal.

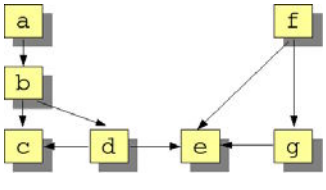


Fig. 1.14. Graphical representation of the **`calls`** relation

Rascal supports basic data types like integers and strings which are sufficient to formulate and answer the questions at hand. However, we can gain readability by introducing separately named types for the items we are describing. First, we introduce therefore a new type `proc` (an alias for strings) to denote procedures:

```
rascal> alias proc = str;
ok
```

Suppose that the following facts have been extracted from the source code and are represented by the relation `Calls`:

```
rascal> rel[proc, proc] Calls =
   { <"a", "b">, <"b", "c">, <"b", "d">, <"d", "c">,
     <"d","e">, <"f", "e">, <"f", "g">, <"g", "e">
   };
rel[proc,proc]: { <"a", "b">, <"b", "c">, <"b", "d">,
                  <"d", "c">, <"d","e">, <"f", "e">,
                  <"f", "g">, <"g", "e">}
```

This concludes the preparatory steps and now we move on to answer the questions.

5.1.2 How Many Procedure Calls Occur in This System?

To determine the numbers of calls, we simply determine the number of tuples in the `Calls` relation, as follows. First, we need the Relation library so we import it:

```
rascal> import Relation;
ok
```

next we describe a new variable and calculate the number of tuples:

```
rascal> nCalls = size(Calls);
int: 8
```

The library function `size` determines the number of elements in a set or relation. In this example, `nCalls` will get the value `8`.

5.1.3 How Many Procedures Are Contained in It?

We get the number of procedures by determining which names occur in (the first or second component of) the tuples in the relation `Calls` and then determining the number of names:

```
rascal> procs = carrier(Calls);
set[proc]: {"a", "b", "c", "d", "e", "f", "g"}

rascal> nprocs = size(procs);
int: 7
```

The built-in function `carrier` determines all the values that occur in the tuples of a relation. In this case, `procs` will get the value `{"a", "b", "c", "d", "e", "f", "g"}` and `nprocs` will thus get value `7`. A more concise way of expressing this would be to combine both steps:

```
rascal> nprocs = size(carrier(Calls));
int: 7
```

5.1.4 What Are the Entry Points for This System?

The next step in the analysis is to determine which *entry points* this application has, i.e., procedures which call others but are not called themselves. Entry points are useful since they define the external interface of a system and may also be used as guidance to split a system in parts. The `top` of a relation contains those left-hand sides of tuples in a relation that do not occur in any right-hand side. When a relation is viewed as a graph, its top corresponds to the root nodes of that graph. Similarly, the `bottom` of a relation corresponds to the leaf nodes of the graph. Using this knowledge, the entry points can be computed by determining the top of the `Calls` relation:

```
rascal> import Graph;
ok

rascal> entryPoints = top(Calls);
set[proc]: {"a", "f"}
```

In this case, `entryPoints` is equal to `{"a", "f"}`. In other words, procedures `"a"` and `"f"` are the entry points of this application.

5.1.5 What Are the Leaves of This Application?

In a similar spirit, we can determine the *leaves* of this application, i.e., procedures that are being called but do not make any calls themselves:

```
rascal> bottomCalls = bottom(Calls);
set[proc]: {"c", "e"}
```

In this case, `bottomCalls` is equal to `{"c", "e"}`.

5.1.6 Which Procedures Call Each Other Indirectly?

We can also determine the *indirect calls* between procedures, by taking the transitive closure of the `Calls` relation, written as `Calls+`. Observe that the transitive closure will contain both the direct and the indirect calls.

```
rascal> closureCalls = Calls+;
rel[proc, proc]: {<"a", "b">, <"b", "c">, <"b", "d">,
                  <"d","e">, <"f", "e">,
                  <"f", "g">, <"g", "e">, <"a", "c">,
                  <"a", "d">, <"b", "e">, <"a", "e">}
```

It is easy to get rid of all the direct calls in the above result, by subtracting (using the set difference operator `-`) the original call relation from it:

```
rascal> indirectClosureCalls = closureCalls - Calls;
rel[proc, proc]: {<"a", "c">, <"a", "d">,
                  <"b", "e">, <"a", "e">}
```

5.1.7 Which Procedures Are Called Directly or Indirectly from Each Entry Point?

We now know the entry points for this application (`"a"` and `"f"`) and the indirect call relations. Combining this information, we can determine which procedures are called from each entry point. This is done by indexing `closureCalls` with an appropriate procedure name. The index operator yields all right-hand sides of tuples that have a given value as left-hand side. This gives the following:

```
rascal> calledFromA = closureCalls["a"];
set[proc]: {"b", "c", "d", "e"}
```

and

```
rascal> calledFromF = closureCalls["f"];
set[proc]: {"e", "g"}
```

5.1.8 Which procedures are called from all entry points?

Finally, we can determine which procedures are called from both entry points by taking the intersection (`&`) of the two sets `calledFromA` and `calledFromF`:

```
rascal> commonProcs = calledFromA & calledFromF;
set[proc]: {"e"}
```

In other words, the procedures called from both entry points are mostly disjoint except for the common procedure `"e"`.

5.1.9 Wrap-Up

These findings can be verified by inspecting a graph view of the calls relation as shown in Figure 1.14, “Graphical representation of the `calls` relation”. Such a visual inspection does *not* scale very well to large graphs and this makes the above form of analysis particularly suited for studying large systems.

This example illustrates the following:

- Call dependencies can easily be represented as relations.
- By design, Rascal contains the full repertoire of operators and library functions for manipulating relations.
- This style of programming enables flexible exploration of given data.

5.2 Analyzing the Component Structure of an Application

A frequently occurring problem is that we know the call relation of a system but that we want to understand it at the component level rather than at the procedure level. If it is known to which component each procedure belongs, it is possible to *lift* the call relation

to the component level as proposed in [Kri99]. Actual lifting amounts to translating each call between procedures to a call between components. This is described in the following module:

```
module demo::Lift

alias proc = str;
alias comp = str;

public rel[comp,comp] lift(rel[proc,proc] aCalls,
                           rel[proc,comp] aPartOf){
   return
      { <C1, C2> | <proc P1, proc P2> <- aCalls,
                   <comp C1, comp C2> <- aPartOf[P1] *
                                         aPartOf[P2]
      };
}
```

For each pair `<P1,P2>` in the Calls relation we compose the corresponding parts `aPartOf[P1]` and `aPartOf[P2]` (each yielding a set of components) into a new relation of calls between components. This relation is added pair by pair to the result.

Let's now apply this. First import the above module, and define a call relation and a partof relation:

```
rascal> import demo::Lift;
ok

rascal> Calls = {<"main", "a">, <"main", "b">, <"a", "b">,
                 <"a", "c">, <"a", "d">, <"b", "d">
                };
rel[str,str] : {<"main", "a">, <"main", "b">, <"a", "b">,
                <"a", "c">, <"a", "d">, <"b", "d">
               }

rascal> Components = {"Appl", "DB", "Lib"};
set[str] : {"Appl", "DB", "Lib"}

rascal> PartOf = {<"main", "Appl">, <"a", "Appl">,
                  <"b", "DB">, <"c", "Lib">,
                  <"d", "Lib">};
rel[str,str] : {<"main", "Appl">, <"a", "Appl">,
                <"b", "DB">, <"c", "Lib">,
                <"d", "Lib">}
```

The lifted call relation between components is now obtained by:

```
rascal> ComponentCalls = lift(Calls, PartOf);
rel[str,str] : {<"DB", "Lib">, <"Appl", "Lib">,
                <"Appl", "DB">, <"Appl", "Appl">}
```

The relevant relations for this example are shown in Figure 1.15, "(a) `Calls`; (b) `PartOf`; (c) `ComponentCalls`.".

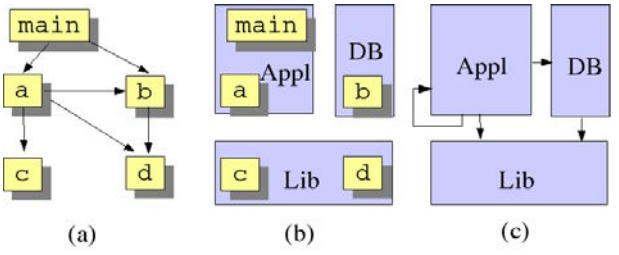


Fig. 1.15. (a) `Calls`; (b) `PartOf`; (c) `ComponentCalls`.

This example illustrates the following:

- A comprehension to build a relation, e.g., as done in `lift`.
- Relation projection, e.g., `aPartOf[P1]`.
- Set product, e.g., `aPartOf[P1] * aPartOf[P2]`.

5.3 Analyzing the Structure of Java Systems

Now we consider the analysis of Java systems (inspired by [BNL05]). Suppose that the type `class` is defined as follows

```
alias class = str;
```

and that the following relations are available about a Java application:

- `rel[class,class] CALL`: If <C_1, C_2> is an element of `CALL`, then some method of C_2 is called from C_1.
- `rel[class,class] INHERITANCE`: If <C_1, C_2> is an element of `INHERITANCE`, then class C_1 either extends class C_2 or C_1 implements interface C_2.
- `rel[class,class] CONTAINMENT`: If <C_1, C_2> is an element of `CONTAINMENT`, then one of the fields of class C_1 is of type C_2.

To make this more explicit, consider the class `LocatorHandle` from the JHotDraw application (version 5.2) as shown here:

```
package CH.ifa.draw.standard;
```

```
import java.awt.Point;
import CH.ifa.draw.framework.*;
/**
 * A LocatorHandle implements a Handle by delegating the
 * location requests to a Locator object.
 */
public class LocatorHandle extends AbstractHandle {
    private Locator       fLocator;
    /**
     * Initializes the LocatorHandle with the
     * given Locator.
     */
    public LocatorHandle(Figure owner, Locator l) {
        super(owner);
        fLocator = l;
    }
    /**
     * Locates the handle on the figure by forwarding
     * the request to its figure.
     */
    public Point locate() {
        return fLocator.locate(owner());
    }
}
```

It leads to the addition to the above relations of the following tuples:

- To `CALL` the pairs `<"LocatorHandle", "AbstractHandle">` and `<"LocatorHandle", "Locator">` will be added.
- To `INHERITANCE` the pair `<"LocatorHandle", "AbstractHandle">` will be added.
- To `CONTAINMENT` the pair `<"LocatorHandle", "Locator">` will be added.

Cyclic structures in object-oriented systems makes understanding hard. Therefore it is interesting to spot classes that occur as part of a cyclic dependency. Here we determine cyclic uses of classes that include calls, inheritance and containment. This is achieved as follows:

```
rel[class,class] USE = CALL + CONTAINMENT + INHERITANCE;
set[str] ClassesInCycle =
   {C1 | <class C1, class C2> <- USE+, C1 == C2};
```

First, we define the `USE` relation as the union of the three available relations `CALL`, `CONTAINMENT` and `INHERITANCE`. Next, we consider all pairs $<C_1, C_2>$ in the

transitive closure of the USE relation such that C_1 and C_2 are equal. Those are precisely the cases of a class with a cyclic dependency on itself. Probably, we want to know not only which classes occur in a cyclic dependency, but we also which classes are involved in such a cycle. In other words, we want to associate with each class a set of classes that are responsible for the cyclic dependency. This can be done as follows.

```
rel[class,class] USE = CALL + CONTAINMENT + INHERITANCE;
set[class] CLASSES = carrier(USE);
rel[class,class] USETRANS = USE+;
rel[class,set[class]] ClassCycles =
   {<C, USETRANS[C]> | class C <- CLASSES,
                       <C, C> in USETRANS };
```

First, we introduce two new shorthands: CLASSES and USETRANS. Next, we consider all classes C with a cyclic dependency and add the pair <C, USETRANS[C]> to the relation ClassCycles. Note that USETRANS[C] is the right image of the relation USETRANS for element C, i.e., all classes that can be called transitively from class C.

This example illustrates the following:

- Representation of facts related to declarations in Java programs.
- Manipulation of these facts using comprehensions and relational operators.

One of the Rascal libraries gives access to the Eclipse JDT and provides information about declarations and types of Java programs in relational form. This is briefly illustrated in Section 5.8, “Visualizing Extracted Information”.

5.4 Finding Uninitialized and Unused Variables in a Program

Consider the following program in the toy language Pico: (This is an extended version of the example presented earlier in [Kli03].)

```
[ 1] begin declare x : natural, y : natural,
[ 2]               z : natural, p : natural;
[ 3]  x := 3;
[ 4]  p := 4;
[ 5]  if q then
[ 6]        z := y + x
[ 7]  else
[ 8]        x := 4
[ 9]  fi;
[10]  y := z
[11] end
```

Inspection of this program shows that some of the variables are being used before they have been initialized. The variables in question are q (line 5), y (line 6), and z (line

10). It is also clear that variable p is initialized (line 4), but is never used. How can we automate these kinds of analysis? Recall from Section 1.1, "The EASY Paradigm" that we follow the Extract-Analyze-SYnthesize paradigm to approach such a problem. The first step is to determine which elementary facts we need about the program. For this and many other kinds of program analysis, we need at least the following:

- The *control flow graph* of the program. We represent it by a graph PRED (for predecessor) which relates each statement with its predecessors.
- The *definitions* of each variable, i.e., the program statements where a value is assigned to the variable. It is represented by the relation DEFS.
- The *uses* of each variable, i.e., the program statements where the value of the variable is used. It is represented by the relation USES.

In this example, we will use line numbers to identify the statements in the program. Assuming that there is a tool to extract the above information from a program text, we get the following for the above example:

```
module demo::Uninit
import Relation;
import Graph;

alias expr = int;
alias varname = str;

public expr ROOT = 1;

public graph[expr] PRED = { <1,3>, <3,4>, <4,5>, <5,6>,
                            <5,8>, <6,10>, <8,10> };

public rel[varname,expr] DEFS = { <"x", 3>, <"p", 4>,
                                  <"z", 6>, <"x", 8>,
                                  <"y", 10> };

public rel[varname, expr] USES = { <"q", 5>, <"y", 6>,
                                   <"x", 6>, <"z", 10> };
```

This concludes the extraction phase. Next, we have to enrich these basic facts to obtain the initialized variables in the program. So, when is a variable *V* in some statement *S* initialized? If we execute the program (starting in ROOT), there may be several possible execution paths that can reach statement *S*. All is well if *all* these execution path contain a definition of *V*. However, if one or more of these path do *not* contain a definition of *V*, then *V* may be uninitialized in statement *S*. This can be formalized as follows:

```
// module demo::Unit continued
public rel[varname,expr] UNINIT =
```

```
   { <V,E> | <varname V, expr E> <- USES,
             E in reachX(PRED, {ROOT}, DEFS[V])
   };
```

We analyze this definition in detail:

- `<varname V, expr E> : USES` enumerates all tuples in the `USES` relation. In other words, we consider the use of each variable in turn.
- `E in reachX(PRED, {ROOT}, DEFS[V])` is a test that determines whether expression *E* is reachable from the `ROOT` without encountering a definition of variable *V*.
  - `PRED` is the relation for which the reachability has to be determined.
  - `{ROOT}` represents the initial set of nodes from which all path should start.
  - `DEFS[V]` yields the set of all statements in which a definition of variable `V` occurs. These nodes form the exclusion set for `reachX`: no path will be extended beyond an element in this set.
  - The result of `reachX(PRED, {ROOT}, DEFS[V])` is a set that contains all nodes that are reachable from the `ROOT`.
  - Finally, `E in reachX(PRED, {ROOT}, DEFS[V])` tests whether expression `E` can be reached from the `ROOT`.
- The net effect is that `UNINIT` will only contain pairs that satisfy the test just described.

When we execute the resulting Rascal code (i.e., the declarations of `ROOT`, `PRED`, `DEFS`, `USES` and `UNINIT`), we get as value for `UNINIT`:

```
rascal> import demo::Uninit;
ok

rascal> UNINIT;
rel[varname,expr]: {<"q", 5>, <"y", 6>, <"z", 10>}
```

and this is in concordance with the informal analysis given at the beginning of this example.

As a bonus, we can also determine the *unused* variables in a program, i.e., variables that are defined but are used nowhere. This is done as follows:

```
// module demo::Unit continued

public set[varname] UNUSED = domain(DEFS) - domain(USES);
```

Taking the domain of the relations `DEFS` and `USES` yields the variables that are defined, respectively, used in the program. The difference of these two sets yields the unused variables, in this case `{"p"}`.

This example illustrates the following:

- Representation of the control flow graph as a relation.
- Reachability computations on that flow graph.

5.5 McCabe Cyclomatic Complexity

The *cyclomatic complexity* of a program is defined as `e - n + 2`, where `e` and `n` are the number of edges and nodes in the control flow graph, respectively. It was proposed by McCabe [McC76] as a measure of program complexity. Experiments have shown that programs with a higher cyclomatic complexity are more difficult to understand and test and have more errors. It is generally accepted that a program, module or procedure with a cyclomatic complexity larger than 15 is *too complex*. Essentially, cyclomatic complexity measures the number of decision points in a program. Given a control flow in the form of a predecessor graph `Graph[&T] PRED` between elements of arbitrary type `&T`, the cyclomatic complexity can be computed in Rascal as follows:

```
module demo::McCabe
import Graph;

public int cyclomaticComplexity(Graph[&T] PRED){
    return size(PRED) - size(carrier(PRED)) + 2;
}
```

The number of edges `e` is equal to the number of tuples in `PRED`. The number of nodes `n` is equal to the number of elements in the carrier of PRED, i.e., all elements that occur in a tuple in `PRED`.

This example illustrates that metrics on a (control flow) graph can be easily expressed.

5.6 Dataflow Analysis

Dataflow analysis is a program analysis technique that forms the basis for many compiler optimizations. It is described in any text book on compiler construction, e.g. [ASU86]. The goal of dataflow analysis is to determine the effect of statements on their surroundings. Typical examples are:

- Dominators (Section 5.6.1, “Dominators”): which nodes in the flow dominate the execution of other nodes?
- Reaching definitions (Section 5.6.2, “Reaching Definitions”): which definitions of variables are still valid at each statement?

- Live variables (Section 5.6.3, "Live Variables"): of which variables will the values be used by successors of a statement?
- Available expressions: which expressions are computed along each path from the start of the program to the current statement?

5.6.1 Dominators

A node d of a flow graph *dominates* a node n, if every path from the initial node of the flow graph to n goes through d [ASU86] (Section 10.4). Dominators play a role in the analysis of conditional statements and loops. The function `dominators` that computes the dominators for a given flow graph `PRED` and an entry node `ROOT` is defined as follows:

```
module demo::Dominators
import Set;
import Relation;
import Graph;

public rel[&T, set[&T]] dominators(rel[&T,&T] PRED,
                                   &T ROOT)
{
  set[&T] VERTICES = carrier(PRED);
  return  { <V,  (VERTICES - {V, ROOT}) -
                 reachX(PRED,{ROOT},{V})>
            |  &T V <- VERTICES
          };
}
```

First, the auxiliary set `VERTICES` (all the statements) is computed. The relation `DOMINATES` consists of all pairs $<S, \{S_1, \ldots, S_n\}>$ such that

- S_i is not an initial node or equal to S.
- S_i cannot be reached from the initial node without going through S.

First import the above module and consider the sample flow graph PRED:

```
rascal> import demo::Dominators;
ok

rascal> rel[int,int] PRED = {
<1,2>, <1,3>,
<2,3>,
<3,4>,
<4,3>,<4,5>, <4,6>,
<5,7>,
```

```
<6,7>,
<7,4>,<7,8>,
<8,9>,<8,10>,<8,3>,
<9,1>,
<10,7>
};

rel[int,int]: { <1,2>, <1,3>, ...
```

It is illustrated inFigure 1.16, “Flow graph”

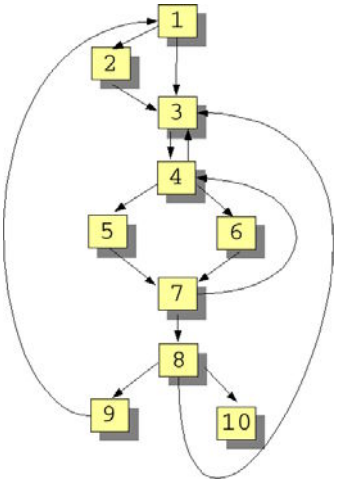


Fig. 1.16. Flow graph

The result of applying `dominators` to it is as follows:

```
rascal> dominators(PRED);
rel[int,int]: {<1, {2, 3, 4, 5, 6, 7, 8, 9, 10}>,
<2, {}>,
<3, {4, 5, 6, 7, 8, 9, 10}>,
<4, {5, 6, 7, 8, 9, 10}>,
<5, {}>,
<6, {}>,
<7, {8, 9, 10}>,
<8, {9, 10}>,
<9, {}>,
<10, {}>}
```

The resulting *dominator tree* is shown in Figure 1.17, “Dominator tree”. The dominator tree has the initial node as root and each node d in the tree only dominates its descendants in the tree.

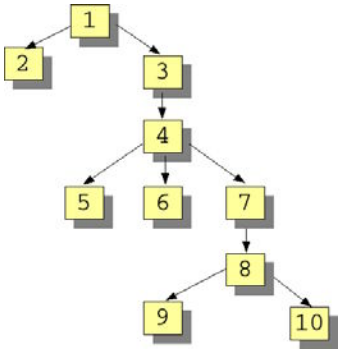


Fig. 1.17. Dominator tree

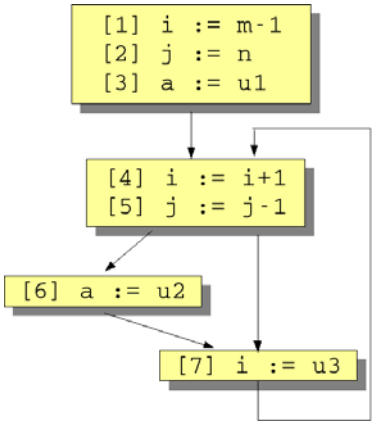


Fig. 1.18. Flow graph for various dataflow problems

5.6.2 Reaching Definitions

We illustrate the calculation of reaching definitions using the example in Figure 1.18, "Flow graph for various dataflow problems" which was inspired by [ASU86] (Example 10.15).

We introduce the notions `definition` and `use` to represent information about the program. The former describes that a certain statement defines some variable and the latter that a statement uses some variable. They are defined as follows:

```
module demo::ReachingDefs

import Relation;
import Graph;
import IO;
```

```
public alias stat = int;
public alias var = str;
public alias def  = tuple[stat, var];
public alias use = tuple[stat,var];

public rel[stat,def] definition(rel[stat,var] DEFS){
  return {<S,<S,V>> | <stat S, var V> <- DEFS};
}

public rel[stat,def] use(rel[stat, var] USES){
  return {<S, <S, V>> | <stat S, var V> <- USES};
}
```

Let's use the following values to represent our example:

```
rascal> rel[stat,stat] PRED = { <1,2>, <2,3>, <3,4>,
                                <4,5>, <5,6>, <5,7>,
                                <6,7>, <7,4> };
rel[stat,stat]: { <1,2>, <2,3>, ...

rascal> rel[stat, var] DEFS = { <1, "i">, <2, "j">,
                                <3, "a">, <4, "i">,
                                <5, "j">, <6, "a">,
                                <7, "i"> };
rel[stat,var]: { <1, "i">, <2, "j">, ...

rascal> rel[stat,var] USES = { <1, "m">, <2, "n">,
                               <3, "u1">, <4, "i">,
                               <5, "j">, <6, "u2">,
                               <7, "u3"> };
rel[stat,var]: { <1, "m">, <2, "n">,...
```

The functions `definition` and `use` have the following effect on our sample data:

```
rascal> definition(DEFS);
rel[stat,def]: { <1, <1, "i">>, <2, <2, "j">>,
                 <3, <3, "a">>, <4, <4, "i">>,
                 <5, <5, "j">>, <6, <6, "a">>,
                 <7, <7, "i">> }

rascal> use(USES);
rel[stat,def]: { <1, <1, "m">>, <2, <2, "n">>,
                 <3, <3, "u1">>, <4, <4, "i">>,
                 <5, <5, "j">>, <6, <6, "u2">>,
                 <7, <7, "u3">> }
```

Now we are ready to define an important new relation KILL. KILL defines which variable definitions are undone (killed) at each statement and is defined by the following function kill:

```
// continuing module demo::ReachingDefs

public rel[stat,def] kill(rel[stat,var] DEFS) {
  return {<S1, <S2, V>> | <stat S1, var V> <- DEFS,
                          <stat S2, V> <- DEFS,
                          S1 != S2};
}
```

In this definition, all variable definitions are compared with each other, and for each variable definition all *other* definitions of the same variable are placed in its kill set. In the example, KILL gets the value

```
rascal> kill(DEFS);
rel[stat,def]:
{ <1, <4, "i">>, <1, <7, "i">>, <2, <5, "j">>,
  <3, <6, "a">>, <4, <1, "i">>, <4, <7, "i">>,
  <5, <2, "j">>, <6, <3, "a">>, <7, <1, "i">>,
  <7, <4, "i">>
}
```

and, for instance, the definition of variable i in statement 1 kills the definitions of i in statements 4 and 7.

After these preparations, we are ready to formulate the reaching definitions problem in terms of two relations IN and OUT. IN captures all the variable definitions that are valid at the entry of each statement and OUT captures the definitions that are still valid after execution of each statement. Intuitively, for each statement S, IN[S] is equal to the union of the OUT of all the predecessors of S. OUT[S], on the other hand, is equal to the definitions generated by S to which we add IN[S] minus the definitions that are killed in S. Mathematically, the following set of equations captures this idea for each statement:

$IN[S] = UNION_{P \text{ in predecessors of } S} OUT[P]$

$OUT[S] = DEF[S] + (IN[S] - KILL[S])$

This idea can be expressed in Rascal quite literally:

```
public rel[stat, def] reachingDefinitions(
                            rel[stat,var] DEFS,
                            rel[stat,stat] PRED){
  set[stat] STATEMENT = carrier(PRED);
  rel[stat,def] DEF  = definition(DEFS);
  rel[stat,def] KILL = kill(DEFS);
```

```
  // The set of mutually recursive dataflow equations
  // that has to be solved:

  rel[stat,def] IN = {};
  rel[stat,def] OUT = DEF;

  solve (IN, OUT) {
    IN  = {<S, D> | int S <- STATEMENT,
                    stat P <- predecessors(PRED,S),
                    def D <- OUT[P]};
    OUT = {<S, D> | int S <- STATEMENT,
                    def D <- DEF[S] + (IN[S] - KILL[S])};
  };
  return IN;
}
```

First, the relations `IN` and `OUT` are declared and initialized. Next follows a solve statement that uses `IN` and `OUT` as variables and contains two equations that resemble the mathematical equations given above. Note the use of the library function `predecessors` to obtain the predecessors of a statement for a given control flow graph.

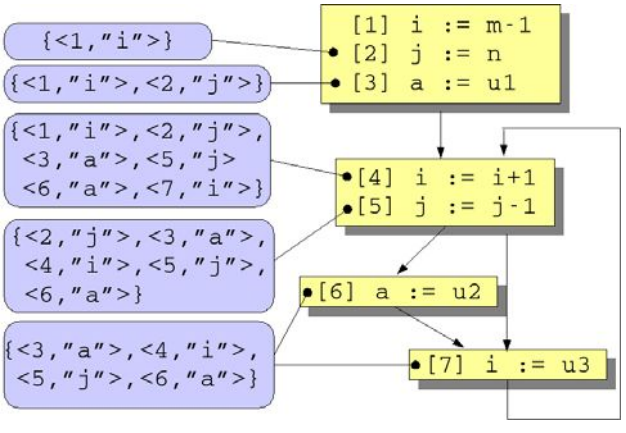


Fig. 1.19. Reaching definitions for example

For our running example the results are as follows (see Figure 1.19, "Reaching definitions for example"). Relation `IN` has as value:

```
{ <2, <1, "i">>, <3, <2, "j">>, <3, <1, "i">>,
  <4, <3, "a">>, <4, <2, "j">>, <4, <1, "i">>,
  <4, <7, "i">>, <4, <5, "j">>, <4, <6, "a">>,
  <5, <4, "i">>, <5, <3, "a">>, <5, <2, "j">>,
  <5, <5, "j">>, <5, <6, "a">>, <6, <5, "j">>,
```

```
  <6, <4, "i">>, <6, <3, "a">>, <6, <6, "a">>,
  <7, <5, "j">>, <7, <4, "i">>, <7, <3, "a">>,
  <7, <6, "a">>
}
```

If we consider statement `3`, then the definitions of variables `i` and `j` from the preceding two statements are still valid. A more interesting case are the definitions that can reach statement `4`:

- The definitions of variables `a`, `j` and `i` from, respectively, statements `3`, `2` and `1`.
- The definition of variable `i` from statement `7` (via the backward control flow path from `7` to `4`).
- The definition of variable `j` from statement `5` (via the path `5`, `7`, `4`).
- The definition of variable `a` from statement `6` (via the path `6`, `7`, `4`).

Relation `OUT` has as value:

```
{ <1, <1, "i">>, <2, <2, "j">>, <2, <1, "i">>,
  <3, <3, "a">>, <3, <2, "j">>, <3, <1, "i">>,
  <4, <4, "i">>, <4, <3, "a">>, <4, <2, "j">>,
  <4, <5, "j">>, <4, <6, "a">>, <5, <5, "j">>,
  <5, <4, "i">>, <5, <3, "a">>, <5, <6, "a">>,
  <6, <6, "a">>, <6, <5, "j">>, <6, <4, "i">>,
  <7, <7, "i">>, <7, <5, "j">>, <7, <3, "a">>,
  <7, <6, "a">>
}
```

Observe, again for statement `4`, that all definitions of variable `i` are missing in `OUT[4]` since they are killed by the definition of `i` in statement `4` itself. Definitions for `a` and `j` are, however, contained in `OUT[4]`. The result of reaching definitions computation is illustrated in Figure 1.19, “Reaching definitions for example”. We will use the function `reachingDefinitions` later on in Section 5.7, “Program Slicing” when defining program slicing.

5.6.3 Live Variables

The live variables of a statement are those variables whose value will be used by the current statement or some successor of it. The mathematical formulation of this problem is as follows:

IN[S] =USE[S] + (OUT[S] - DEF[S])

OUT[S] = UNION$_{S' \text{ in successors of } S}$ IN[S']

The first equation says that a variable is live coming into a statement if either it is used before redefinition in that statement or it is live coming out of the statement and is not

redefined in it. The second equation says that a variable is live coming out of a statement if and only if it is live coming into one of its successors.

This can be expressed in Rascal as follows:

```
public rel[stat,def] liveVariables(rel[stat, var] DEFS,
                                   rel[stat, var] USES,
                                   rel[stat,stat] PRED){
  set[stat] STATEMENT = carrier(PRED);
  rel[stat,def] DEF  = definition(DEFS);
  rel[stat,def] USE = use(USES);

  rel[stat,def] LIN = {};
  rel[stat,def] LOUT = DEF;

  solve(LIN, LOUT) {
    LIN  = { <S, D> | stat S <- STATEMENT,
                      def D <- USE[S] +
                               (LOUT[S] - (DEF[S]))};
    LOUT = { <S, D> | stat S <- STATEMENT,
                      stat Succ <- successors(PRED,S),
                      def D <- LIN[Succ] };
  }
  return LIN;
}
```

The results of live variable analysis for our running example are illustrated in Figure 1.20, “Live variables for example”.

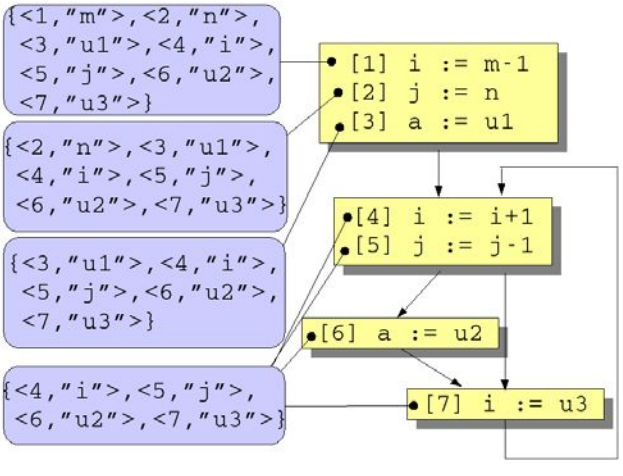


Fig. 1.20. Live variables for example

5.6.4 Wrap Up

The examples in Section 5.6, "Dataflow Analysis" illustrate the following:

- The use of relations to represent control flow and dataflow graphs.
- The use of the solve statement to perform fixed point computations (in these examples to solve dataflow equations).

5.7 Program Slicing

Program slicing is a technique proposed by Weiser [Wei84] for automatically decomposing programs in parts by analyzing their data flow and control flow. Typically, a given statement in a program is selected as the *slicing criterion* and the original program is reduced to an independent subprogram, called a *slice*, that is guaranteed to represent faithfully the behavior of the original program at the slicing criterion. An example will illustrate this (we use line numbers for later reference):

```
[ 1] read(n)          [1] read(n)        [ 1] read(n)
[ 2] i := 1           [2] i := 1         [ 2] i := 1
[ 3] sum := 0         [3] sum := 0
[ 4] product := 1                        [ 4] product := 1
[ 5] while i<= n      [5] while i<= n    [ 5] while i<= n
     do                    do                 do
     begin                 begin              begin
[ 6]  sum :=          [6]  sum :=
          sum + i               sum + i
[ 7]  product :=                         [ 7]  product :=
        product * i                              product * i
[ 8]  i := i + 1      [8]  i := i + 1    [ 8]  i := i + 1
     end                   end                end
[ 9] write(sum)       [9] write(sum)
[10] write(product)                      [10] write(product)

(a) Sample program    (b) Slice for      (c) Slice for
                          statement [9]      statement [10]
```

The initial program is given as (a). The slice with statement [9] as slicing criterion is shown in (b): statements `[4]` and `[7]` are irrelevant for computing statement `[9]` and do not occur in the slice. Similarly, (c) shows the slice with statement `[10]` as slicing criterion. This particular form of slicing is called *backward slicing*. Slicing can be used for debugging and program understanding, optimization and more. An overview of slicing techniques and applications can be found in [Tip95]. Here we will explore a relational formulation of slicing adapted from a proposal in [JR94]. The basic ingredients of the approach are as follows:

- We assume the relations `PRED`, `DEFS` and `USES` as before.

- We assume an additional set `CONTROL-STATEMENT` that defines which statements are control statements.
- To tie together dataflow and control flow, three auxiliary variables are introduced:
  - The variable `TEST` represents the outcome of a specific test of some conditional statement. The conditional statement defines `TEST` and all statements that are control dependent on this conditional statement will use `TEST`.
  - The variable `EXEC` represents the potential execution dependence of a statement on some conditional statement. The dependent statement defines `EXEC` and an explicit (control) dependence is made between `EXEC` and the corresponding `TEST`.
  - The variable `CONST` represents an arbitrary constant.

The calculation of a (backward) slice now proceeds in six steps:

- Compute the relation `rel[use,def] use-def` that relates all uses to their corresponding definitions. The function `reaching-definitions` shown earlier in Section 5.6.2, "Reaching Definitions" does most of the work.
- Compute the relation `rel[def,use] def-use-per-stat` that relates the *internal* definitions and uses of a statement.
- Compute the relation `rel[def,use] control-dependence` that links all `EXEC`s to the corresponding `TEST`s.
- Compute the relation `rel[use,def] use-control-def` that combines use/def dependencies with control dependencies.
- After these preparations, compute the relation `rel[use,use] USE-USE` that contains dependencies of uses on uses.
- The backward slice for a given slicing criterion (a use) is now simply the projection of `USE-USE` for the slicing criterion.

This informal description of backward slicing can now be expressed in Rascal:

```
module demo::Slicing

import Set;
import Relation;
import demo::ReachingDefs;
import demo::Dominators;
import UnitTest;

set[use] BackwardSlice(set[stat] CONTROLSTATEMENT,
                       rel[stat,stat] PRED,
                       rel[stat,var] USES,
```

```
                        rel[stat,var] DEFS,
                        use Criterion) {

  rel[stat, def] REACH = reachingDefinitions(DEFS, PRED);

  // Compute the relation between each use and
  // corresponding definitions: use_def

  rel[use,def] use_def =
  {<<S1,V>, <S2,V>> | <stat S1, var V> <- USES,
                      <stat S2, V> <- REACH[S1]};

  // Internal dependencies per statement

  rel[def,use] def_use_per_stat  =
       {<<S,V1>, <S,V2>> | <stat S, var V1> <- DEFS,
                           <S, var V2> <- USES}
       +
       {<<S,V>, <S,"EXEC">> | <stat S, var V> <- DEFS}
       +
       {<<S,"TEST">,<S,V>> | stat S <- CONTROLSTATEMENT,
                             <S, var V> <-
                                      domainR(USES, {S})};

  // Control dependence: control-dependence

  rel[stat, set[stat]] CONTROLDOMINATOR =
  domainR(dominators(PRED, 1), CONTROLSTATEMENT);

  rel[def,use] control_dependence  =
  { <<S2, "EXEC">,<S1,"TEST">>
    | <stat S1, stat S2> <- CONTROLDOMINATOR};

  // Control and data dependence: use-control-def

  rel[use,def] use_control_def =
               use_def + control_dependence;
  rel[use,use] USE_USE =
               (use_control_def o def_use_per_stat)*;

  return USE_USE[Criterion];
}
```

Let's apply this to the example from the start of this section and assume the following:

```
rascal> import demo::Slicing;
```

```
ok

rascal> rel[stat,stat] PRED = { <1,2>, <2,3>, <3,4>,
                                <4,5>, <5,6>, <5,9>,
                                <6,7>, <7,8>, <8,5>,
                                <8,9>, <9,10> };
rel[stat,stat]: {<1,2>, ...

rascal> rel[stat,var] DEFS  = { <1, "n">, <2, "i">,
                                <3, "sum">,
                                <4,"product">,
                                <6, "sum">,
                                <7, "product">,
                                <8, "i"> };
rel[stat,var]: {<1, "n">, ...

rascal> rel[stat,var] USES  = { <5, "i">, <5, "n">,
                                <6, "sum">, <6,"i">,
                                <7, "product">, <7, "i">,
                                <8, "i">, <9, "sum">,
                                <10, "product">
                              };
rel[stat,var]; { <5, "i"> ...

rascal> set[int] CONTROL-STATEMENT = { 5 };
set[int]: {5}

rascal> BackwardSlice(CONTROL-STATEMENT,
                      PRED, USES, DEFS, <9, "sum">);
set[use]: { <1, "EXEC">, <2, "EXEC">,  <3, "EXEC">,
            <5, "i">, <5, "n">, <6, "sum">, <6, "i">,
            <6, "EXEC">, <8, "i">, <8, "EXEC">,
            <9, "sum"> }
```

Taking the domain of this result, we get exactly the statements in (b) of the example.

This example illustrates once more the use of relations as the basis for program analysis.

5.8 Visualizing Extracted Information

We are now interested in extracting information from Java source code and in visualizing this information. More precisely:

- Given are Java source files that have been imported to Eclipse as an Eclipse project.
- Extract all class declarations from the sources.

- For each class declaration extract:
  - The number of interfaces implemented by the class.
  - The number of attributes (fields) declared in the class.
  - The number of methods declared in the class.
- Visualize this information as follows:
  - Each class is represented by a rectangle.
  - The color of the rectangle represents the number of implemented interfaces.
  - The width of the rectangle represents the number of fields.
  - The height of the rectangle represents the number of methods.

The Rascal program to achive this is as follows:

```
module Metrics

import vis::Render;
import vis::Figure;
import Resources;
import JDT;
import Java;
import Set;
import Map;
import IO;

public Figure metrics(loc project) {
  println("Extracting facts from <project>:");
  facts = extractProject(project);
  return visualize(facts);
}

private FProperty popup(str S){
  return mouseOver(box([fillColor("yellow")],
                   text([fontSize(15),
                         fontColor("black")], S)));
}

private Figure visualize(Resource facts) {
  classes =
  {c | c:entity([_*,class(_)]) <- facts@declaredTopTypes};
```

```
  fields =
  (e : size((facts@declaredFields)[e]) | e <- classes);

  methods =
  (e : size((facts@declaredMethods)[e]) | e <- classes);

  ifaces =
  (e : size((facts@implements)[e]) | e <- classes);

  sizes =   (e : l.length | <l,e> <- facts@types);

  sc = colorScale(toList(ifaces<1>),
                  color("grey"), color("red"));
  println("<size(classes)> classes\n");

  println("Creating visualization ...");
  return pack([size(300, 300),gap(10),center()],
              [box([width(2*fields[e]),
                    height(2*methods[e]),
                    fillColor(sc(ifaces[e])),
                    popup("<readable(e)>
                           Fields:<fields[e]>;
                           Methods:<methods[e]>;
                           Interfaces:<ifaces[e]>")
                    ])
               | e <- classes]);
}

public void main(){
  render(metrics(|project://org.eclipse.imp.pdb.values|));
}
```

It consist of the following parts:

- First relevant libraries are imported, in particular those for visualisation and Java fact extraction.
- Next, the function `metrics` is declared: it takes a location of a Java project, extracts the facts for this project and calls `visualize` to do the actual visualization.
- The function popup produces a text popup to be used in the visualization (it will appear when hovering over a figure).
- The heavy lifting is done in the function `visualize`. First, the facts are massaged in a form suitable for display, and then a list of boxes is created with the visual properties corrresponding to each class. FInally, these boxes are packed together to achieve a minimal display area for the visualization.

- Function `main`, finally, calls `metrics` for a specific project and renders the resulting figure.

The resulting visualization, embedded in the Rascal IDE, is shown in Figure 1.21, “Rascal IDE showing the Metrics visualization”.

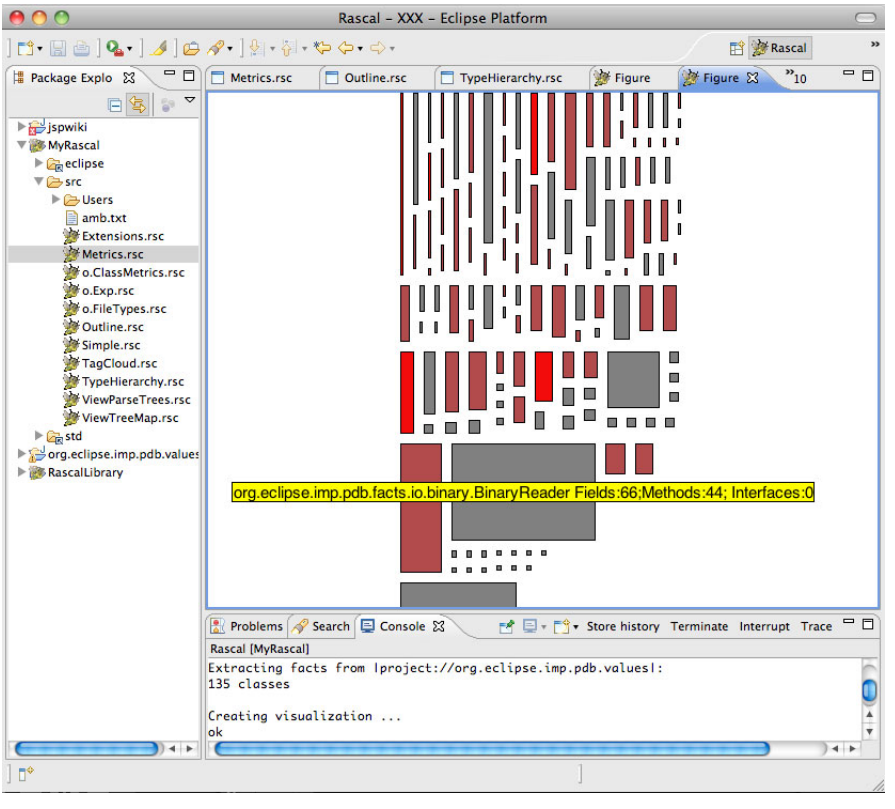


Fig. 1.21. Rascal IDE showing the Metrics visualization

This example illustrates the following:

- Use of the JDT library for fact extraction from Java source code.
- Use of the Figure library for visualization.
- Rascal as bridge between fact extraction and visualization.

6 Concluding Remarks

Rascal and its IDE are in full development at the time of writing and a prototype implementation is available for download. We have given here only a sketch of the language and its applications. The following topics have not been covered in this article:

- The use of SDF modules to parse source text.
- The extensive Rascal library that supports many operations on basic data types including shortest path, reachability and bisimulation on graphs. It also provides tools for drawing graphs and charts, for extracting data from Subversion repositories and more. As mentioned before, our design goal is that Rascal and supporting libraries and tools give as much support as possible for the EASY workflow. This means that we want to provide as much functionality as possible for extraction, analysis, and synthesis. Currently, we provide access to various data and repository formats (XML, HTML, RSF, CVS, SVN, GIT). The Rascal Eclipse JDT library provides direct access to facts that have been extracted from Java source code. The Figure library provides visualization tools. Since the number potentially relevant technologies is unlimited we cannot strive for any form of compleness, but we extend the library on a call by need basis.
- The Rascal IDE that is based on Eclipse and that provides, for instance, very good interactive debugging facilities.

We refer the interested reader to http://www.rascal-mpl.org/ for a more complete and up-to-date overview and for downloading the latest version of Rascal that extends and enhances Rascal version 0.1 as described here.

The main contribution of this work is providing a language, libraries and tools that make it simpler to carry out software engineering tasks that fit the EASY paradigm. The Rascal language is based on many known concepts but adds some innovations as well: the deep integration of grammars (non-terminals are types; parsers are generated on the fly), integrated pattern matching for regular expressions, abstract and concrete syntactic patterns, rewrite rules as normalization device for structured values, and the visit statement for expressing tree visits illustrate this. We have designed Rascal as the glue that can combine the diverse technologies that are needed to carry out a range of tasks in the domain of software analysis and transformation.

Acknowledgements

Rascal has been designed and implemented by the authors but they have received strong support, encouragement, and help from the following individuals. We are very grateful to them.

Emilie Balland implemented the Rascal debugger in the Eclipse version of Rascal during her visit to CWI in the summer of 2009. Bas Basten provided useful feedback on the design and is developing a Rascal/Eclipse JDT interface that makes it easy to extract facts from Java source code. Joppe Kroon further improved the JDT interface.

Bob Fuhrer's inspiring IMP project motivated us to build Rascal on top of IMP. The PDB was designed and implemented during Jurgen's visit to IBM Research in 2007-2008.

Arnold Lankamp implemented a very efficient version of the Program Data Base (PDB), added a binary streaming format, implemented the SGLR invoker, and currently takes

care of deployment issues. Jeroen van den Bos implemented a Rascal library to access subversion repositories. Mark Hills implemented the dataTime data type as well as the static typechecker for Rascal.

Claus Brabrand, Karel Pieterson, Frank Tip and Yaroslav Usenko provided feedback on this paper and suggested several improvements.

We thank the anonymous reviewers for their suggestions for improving this paper.

References

[ASU86] Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley, Reading (1986)

[AZ05] Anderson, P., Zarins, M.: The CodeSurfer software understanding platform. In: Proceedings of the 13th International Workshop on Program Comprehension (IWPC 2005), pp. 147–148. IEEE, Los Alamitos (2005)

[BBC+10] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri- Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: using static analysis to find bugs in the real world. ACM Commun. 53(2), 66–75 (2010)

[BKK+98] Borovansky, P., Kirchner, C., Kirchner, H., Moreau, P.-E., Ringeissen, C.: An overview of ELAN. In: Kirchner, C., Kirchner, H. (eds.) Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications (WRLA 1998). Electronic Notes in Theoretical Computer Science, vol. 15, pp. 55–70 (1998)

[BBK+07] Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E., Reilles, A.: Tom: Piggybacking rewriting on Java. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 36–47. Springer, Heidelberg (2007)

[BPM04] Baxter, I., Pidgeon, P., Mehlich, M.: DMS: Program transformations for practical scalable software evolution. In: Proceedings of the International Conference on Software Engineering (ICSE 2004), pp. 625–634. IEEE, Los Alamitos (2004)

[BDH+01] van den Brand, M.G.J., van Deursen, A., Heering, J., de Jong, H.A., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 365–370. Springer, Heidelberg (2001)

[Bey06] Beyer, D.: Relational programming with crocopat. In: Proceedings of the 28th International Conference on Software Engineering, ICSE 2006, pp. 807–810. ACM, New York (2006)

[BNL05] Beyer, D., Noack, A., Lewerentz, C.: Efficient relational calculation for software analysis. IEEE Trans. Software Engineering 31(2), 137–149 (2005)

[BKV03] van den Brand, M.G.J., Klint, P., Vinju, J.J.: Term rewriting with traversal functions. ACM Transactions on Software Engineering Methodology 12(2), 152–190 (2003)

[BKVV08] Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/ XT 0.17. A language and toolset for program transformation. Science of Computer Programming 72(1-2), 52–70 (2008)

[DN04] Dams, D., Namjoshi, K.S.: Orion: High-precision methods for static error analysis of C and C++ programs. Technical report. Bell Labs, Bell Labs Technical Memorandum ITD-04-45263Z (2004)

[Cor06] Cordy, J.R.: The TXL source transformation language. Science of Computer Programming 61(3), 190–210 (2006)

[FKO98] Feijs, L.M.G., Krikhaar, R., Ommering, R.C.: A relational approach to support software architecture analysis. Software Practice and Experience 28(4), 371–400 (1998)

[Hol08] Holt, R.C.: Grokking software architecture. In: Proceedings of the 15th Working Conference on Reverse Engineering (WCRE 2008), pp. 5–14. IEEE, Los Alamitos (2008)

[HM03] Hedin, G., Magnusson, E.: The JastAdd system - an aspect-oriented compiler construction system. Science of Computer Programming, 37–58 (2003)

[Joh79] Johnson, S.C.: Lint, a program checker. In: McIlroy, M.D., Kernighan, B.W. (eds.) Unix Programmer's Manual, 7th edn., vol. 2B. AT&T Bell Laboratories, Murray Hill (1979)

[JPJ+90] Jourdan, M., Parigot, D., Julié, C., Durin, O., Le Bellec, C.: Design, implementation and evaluation of the FNC-2 attribute grammar system. In: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, PLDI 1990, pp. 209–222. ACM, New York (1990)

[JR94] Jackson, D.J., Rollins, E.J.: A new model of program dependences for reverse engineering. In: Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering. ACM SIGSOFT Software Engineering Notes, vol. 19, pp. 2–10 (1994)

[KHR07] Kniesel, G., Hannemann, J., Rho, T.: A comparison of logic-based infrastructures for concern detection and extraction. In: Proceedings of the 3rd Workshop on Linking Aspect Technology and Evolution, LATE 2007, p. 6. ACM, New York (2007)

[Kli93] Klint, P.: A meta-environment for generating programming environments. ACM Transactions on Software Engineering and Methodology 2(2), 176–201 (1993)

[Kli03] Klint, P.: How understanding and restructuring differ from compiling—a rewriting perspective. In: Proceedings of the 11th International Workshop on Program Comprehension (IWPC 2003), pp. 2–12. IEEE Computer Society, Los Alamitos (2003)

[Kli08] Klint, P.: Using Rscript for software analysis. In: Working Session on Query Technologies and Applications for Program Comprehension, QTAPC 2008 (2008)

[KvdSV09] Klint, P., van der Storm, T., Vinju, J.J.: RASCAL: A domain specific language for source code analysis and manipulation. In: IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2009), pp. 168–177. IEEE Computer Society, Los Alamitos (2009)

[KvdSV10] Klint, P., van der Storm, T., Vinju, J.: DSL tools: Less maintenance? In: Preliminary Proceedings of the Tenth Workshop on Language Descriptions Tools and Applications LDTA 2010, March 27-28 (2010)

[Kri99] Krikhaar, R.L.: Software Architecture Reconstruction. PhD thesis. University of Amsterdam (1999)

[LR01] Lämmel, R., Riedewald, G.: Prological Language Processing. In: Proceedings of the First Workshop on Language Descriptions, Tools and Applications (LDTA 2001), Genova, Italy, April 7 (2001); van den Brand, M., Parigot, D.: Satellite event of ETAPS 2001. ENTCS, vol. 44. Elsevier Science, Amsterdam (April 2001)

[McC76] McCabe, T.J.: A complexity measure. IEEE Transactions on Software Engineering SE-12(3), 308–320 (1976)

[dMSV+08] de Moor, O., Sereni, D., Verbaere, M., Hajiyev, E., Avgustinov, P., Ekman, T., Ongkingco, N., Tibble, J.: QL: Object-oriented queries made easy. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 78–133. Springer, Heidelberg (2008)

[MK88] Müller, H., Klashinsky, K.: Rigi – a system for programming-in-the-large. In: Proceedings of the 10th International Conference on Software Engineering (ICSE 10), pp. 80–86 (April 1988)

[Par07] Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf (2007)

[SRK07] Speicher, D., Rho, T., Kniesel, G.: Jtransformer - eine logikbasierte infrastruktur zur codeanalyse. In: Workshop Software-Reengineering, WSR 2007, May 02-04 (2007)

[Tip95] Tip, F.: A survey of program slicing techniques. Journal of Programming Languages 3(3), 121–189 (1995)

[Wei84] Weiser, M.: Program slicing. IEEE Transactions on Software Engineering SE-10(4), 352–357 (1984)

[VWBGK10] Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: An extensible attribute grammar system. Sci. Comput. Program. 75(1-2), 39–54 (2010)

The Theory and Practice of Modeling Language Design for Model-Based Software Engineering—A Personal Perspective

Bran Selic

Malina Software Corp. Nepean, Ontario, Canada
selic@acm.org

Abstract. The design of modeling languages is still much more of an art than a science. There is as yet no systematic consolidated body of knowledge that a practitioner can refer to when designing a computer-based modeling language. This overview article provides a personal perspective, comprising a selective summary of some important lessons learned and experiences gained in the design of some of the currently most widely used modeling languages, in particular the industry standard UML and MOF languages. The purpose is to provide readers with a sense of the state of the practice and state of the theory, such as it is, based on the author's long-term experience in this domain. Various key concepts involved are defined, current common methods of language design are explored, and heuristic guidelines provided. A list of key research topics is included at the end.

Keywords: engineering models, modeling languages, model-based engineering, model-driven development, computer language design, metamodeling, MOF, EMF, UML, profiles, programming language semantics.

1 Introduction

Modeling is an age-old human activity whereby an artifact is constructed that resembles in some way an imagined or existing system or process. Models serve as surrogates of the system or process that they represent in order to help us understand or appreciate it more. And, although they come in many different forms and are used for many different purposes, some practical and some less so, in almost all cases, the intent of modeling is to *reduce* the full scale of the represented phenomenon to something accessible to human comprehension or some type of formal treatment. The link between models, modeling, and human understanding is often overlooked, but it is crucial if we are to understand how to construct useful models and modeling languages.

Of particular interest to us here are *engineering models*, that is, models used in the analysis, design, and construction of engineering artifacts. Engineers use models for four primary purposes:

1. Models help us understand the represented phenomenon. This is especially useful when the complexity of the phenomenon is such that it challenges our cognitive capacities.

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 290–321, 2011.

2. Models also help us communicate our understanding to others.
3. Models are often used to predict some important characteristics of the represented phenomena.
4. Last but not least, engineering models are often used as blueprints, that is, design specifications that guide implementations.

Since software-based systems are among the most complex systems built by man, it is to be expected that models play a particularly critical role in software development, especially since the model and its corresponding software share the same medium and can, therefore, be formally related to each other through automated transformations and hyperlinks (traces). In principle, this can help reduce the likelihood that an implementation will diverge significantly from its specification. (Strange as it may seem, the significance of models in software development is still highly contentious. For one analysis of this issue refer to [14].)

A *software model* is an engineering model of some software along with its related artifacts (such as its environment, requirements, etc.). Software models are usually described using one or more *modeling languages*. In general, modeling languages do not have to be formal or even computer based—many software models are expressed using natural languages. However, in this article, we are only interested in those software modeling languages that can be processed by a computer in some way and will only discuss those in the remainder of this document. Note that even though a modeling language may be computer based, it does not necessarily follow that the language is precise, formal, or executable. In fact, the vast majority of computer-based modeling languages in use are informal, designed primarily for documentation purposes.

Unfortunately, as can be expected, informal languages tend to be imprecise and ambiguous, opening up the possibility of misinterpretation of the model, which, because it is often difficult to detect, can lead to significant but not always obvious differences between design intent expressed though a model and its actual realization. Furthermore, if such a model is used as a blueprint, it has the additional drawback of not being able to help us predict with confidence whether or not the proposed design is adequate (or even feasible). As Bertrand Meyer noted in a tongue-in-cheek article commenting on the then newly-revealed Unified Modeling Language (UML): "...bubbles and arrows,...as opposed to programs, never crash" [7]. These issues have proven quite troublesome and have led to a rather skeptical attitude about the benefits of modeling among many software developers. For example, adherents of the so-called "agile" movement in software engineering often reject any serious use of modeling, claiming that it is counterproductive.

Still, the movement towards modeling of complex software systems seems inevitable, since models, being abstractions, are the only truly effective means by which humans can cope with the sheer complexity of much modern software. Therefore, to make modeling more effective, it is necessary that we design good and useful modeling languages. Unfortunately, although modeling languages have been around since the dawn of computing (consider, for instance, the classical flow chart), it is only in the past few decades that the topic of modeling language design has received due attention. This means that we are still unsure of how to design these languages, what distinguishes a good one from a bad one and why, what is the proper process of designing such a language, and so on. Consequently, it might have been more appropriate to have put the

word "theory" in the title of this article between quotation marks, as such theory that is currently present is fragmented, incomplete, and sometimes inconsistent. Current approaches are mostly based on heuristics, some of which are documented in this article. These heuristics come mostly from industrial experience, although the pace of research is accelerating. Given the lack of theoretical underpinnings, it is rather surprising how far industry has managed to progress in the definition and use of modeling languages to practical ends. There have been numerous successes with the application of modeling languages and methods in large industrial projects (c.f., [9][16]). Yet, it is difficult to reproduce these successes across the board. In the absence of a consolidated general theory of modeling language design and use, there is no guarantee that what may have worked on one project or system will necessarily succeed in other cases.

The purpose of this overview article is to document some of the author's personal experiences and lessons learned, gained from extensive and direct involvement in the design and implementation of a number of significant modeling languages, both so-called general purpose languages, such as UML [12], as well as domain-specific languages, such as MOF [10] and ROOM [13]. While an attempt has been made to organize this knowledge into some semblance of order, it is far from being a systematic and comprehensive review of either the state of the art or practice. Rather, as the title forewarns, it captures a *personal perspective*. There are now available other references that deal with the topic of modeling language design, including notably [5][4][2][6]. However, the different viewpoints expressed in these sources clearly and accurately illustrate the lack of a consensus and the immaturity of the technical discipline. Consequently, readers are cautioned to treat this overview as yet another perspective on the problem of modeling language design. I contribute it in the hope that it will prove useful to both its readers as well as those who are hoping to consolidate the various views and make sense of it all.

Section 2 starts with a definition of what I feel constitutes a computer-based modeling language and its principal components. This is then used as the basis for structuring the rest of the paper, except for a brief digression on the rather controversial topic (needlessly so, in my view) of general-purpose versus domain-specific languages in section 3. Sections 4 (Metamodeling), 5 (Profile-Based DSML Definition) and 6 (A Systematic Method for Defining UML Profiles) deal with the methods for defining the abstract syntax of modeling languages. Special attention is placed on the UML profile mechanism and its use, which, unfortunately, have not received due attention from a theoretical point of view – although it is used quite extensively in practice. Section 7 focuses on semantics, their design and specification. Finally, some conclusions and a list of major research topics are discussed in section 8.

One highly relevant topic that is intentionally omitted here, albeit reluctantly, is the issue of model transformations, both model-to-model and model-to-code. As experience with programming languages has shown, this is an important factor that can have significant effect in the design of a computer language. However, its inclusion would have greatly expanded the scope and length of this overview.

2 The Key Elements of Computer Modeling Languages

The three key components of the majority of current modeling languages and their principal relationships are shown in the UML class diagram in Fig. 1.

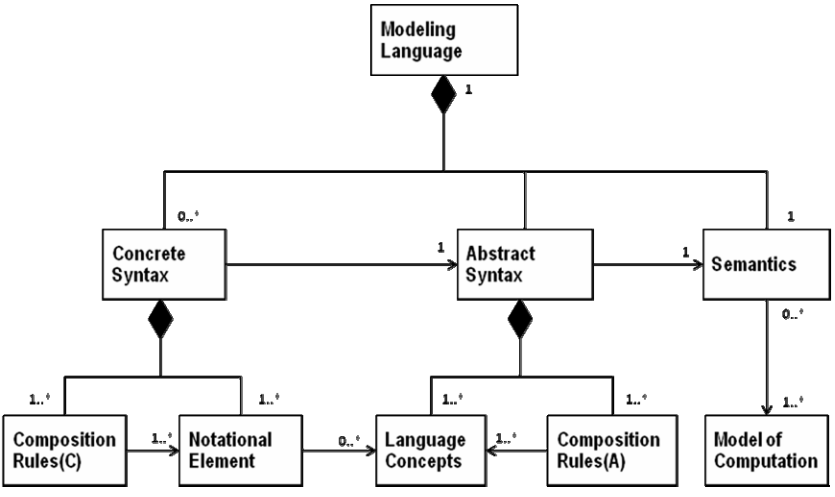


Fig. 1. Elements of a modeling language specification

The *abstract syntax* defines the set of *language concepts* and the *composition rules* that represent the "algebra" for combining these concepts into valid or so-called "well-formed" models. This syntax is called “abstract” to distinguish it from the *concrete syntax*, that is, the actual human-readable notation used to present and view models. Although it is not necessarily the case, most present-day modeling languages keep the abstract syntax separate from and independent of the concrete syntax.

Distinguishing these two kinds of syntaxes opens up the possibility for a given modeling language to have multiple different concrete representations (e.g., depending on the context or the viewpoint chosen). This idea is certainly not new, but it has been exploited much more in modeling language design compared to traditional programming languages, due to the much greater emphasis on the communication function of models. The latter is the primary reason why so many modeling languages opt for a graphical syntax, since graphics are often significantly more conducive to human understanding than text (e.g., finite state machines are typically more easily comprehended as graphs than as text). In other words, in modeling language design the communication value of models is deemed a first-order design concern. In fact, this additional focus may be the primary differentiator between programming languages (as represented by the current set of widely-used languages such as Java or C#) and current modeling languages.

As an example, UML has both graphical and textual concrete syntaxes. Like the abstract syntax, the concrete syntax consists of a set of *notational elements* and its own set of *composition rules*. However, in general, there is not always a direct one-to-one correspondence between the two syntaxes. For example, a given abstract syntax concept may have more than one representation within a given concrete syntax, or, a single notational element may represent multiple language concepts. Some language concepts may capture language abstractions and, thus, may not even have a notational equivalent. Conversely, some notational elements may not have a corresponding abstract syntax element (e.g., punctuation symbols to delineate tokens in a textual concrete syntax).

As an example, assume that we are asked to define a modeling language for describing the game of football (soccer). The abstract syntax of our language is likely to include concepts such as Player, Coach, Game, Team, Goal, Ball, etc. In addition, it might include as part of the abstract syntax various relationships that must hold between these concepts, such as the fact that a Team consists of Players and Coaches or that a Game involves two Teams. In addition, these may be supplemented by various rules and constraints, such as that, in a valid Game, each Team must field a minimum of 7 players and a maximum of 11 players. Not all concepts included in the abstract syntax are necessarily directly available for use by modelers. For ease of language definition and evolution, it is sometimes convenient to include *abstract concepts* in the abstract syntax. These concepts do not have a corresponding concrete representation and, therefore, cannot appear in a model. For instance, we might introduce an abstract concept such as Person to capture features that are common to both Players and Coaches. The latter can then be defined more simply as specializations of the abstract concept. To distinguish those concepts that can be specified by modelers, we shall refer to them specifically as *constructs,* since they are directly available for constructing models.

For our concrete syntax, we may choose to use a graphical approach, in which, for example, we may choose to represent Players using icons representing human soccer players, Teams by icons representing their team logos with arcs connecting them to those Players belonging to the team, etc. Or, we may define a textual syntax using keywords such as "**`player`**" and "**`team`**" to represent the language concepts.

The third major component of the definition of a modeling language (or any computer language for that matter) is a specification of its *semantics*. Simply put, the semantics defines the meaning of the concepts of a language. In case of modeling languages, which are used to specify models of the real-world (software, people, machines, actions, etc.), the semantics define the real-world entity or phenomenon that each language concept represents (e.g., the Player concept in our example would be the computer representation of some individual who plays football on a team). There are many different ways of specifying semantics of computer languages, both formal and informal. Since we are solely interested in computer modeling languages here, a key element in the definition of the semantics of a language is selecting the *model of computation*. This is a conceptual model of how the computation that is being modeled actually occurs. Many different models of computation have emerged over time in computer science, including algorithmic models, event-driven models, a flow-based model, logic programming models, etc.

There are numerous inter-dependent design choices that have to be made when designing a modeling language. The following is a sample set of top-level questions to be addressed when designing a modeling language:

- Should the language be domain specific or general purpose?
- Is the language to be used primarily for documentation or will it also support implementation?
- Should the language be defined informally or formally?
- Should the language be executable or not?
- What should be the dominant model of computation behind the language?
- Should the language include facilities for extension?

- Should the language be designed from scratch or as a refinement of an existing modeling language (e.g., as a profile)?

In designing the abstract syntax, some crucial questions that need to be addressed are:

- What approach should be used to define the language concepts and their relationships (e.g., meta-modeling or BNF)?
- How should well-formedness rules and constraints be defined?
- Should the abstract syntax specification take advantage of generalization mechanisms?

When it comes to defining a concrete syntax, top-level questions that need corresponding design choices include:

- Should the language have a graphical, textual, or combined syntax?
- What rules and guidelines should be used to guarantee consistency of syntax?
- Should it be possible to support multiple representations of the same element?
- Should the language support multiple viewpoints?
- How should the concrete syntax be specified? (Note that, in case of graphical languages, there is no satisfactory agreed on method for specifying a notation)
- How should the mapping from the concrete syntax to the abstract syntax be specified?
- Etc.

Finally, related to semantics, the following are some key questions:

- What method of specifying semantics should be used (operational, denotation, axiomatic, natural language, etc.)?
- If multiple models of computation are used, how are they reconciled with each other?

Unfortunately, the state of the art of modeling language design is such that we do not yet have a systematic and comprehensive set of answers to many of these questions. There is no established body of theory or time-proven guidelines on which to rely to ensure that we produce a reasonably usable and consistent language for a given set of requirements. In what follows, we will discuss some putative answers to some of these questions, but certainly not all, partly because we are not yet sure of how to answer them best in given circumstances.

3 Domain-Specific Modeling Languages

Specialization is a relentless ever-increasing trend in all facets of society, including software. We are seeing a constant branching of knowledge domains into yet more refined and specialized sub-domains and so on. For example, the general discipline of engineering started off as a more or less unified body of knowledge (c.f., [15]), eventually branching out into various specialties (such as mechanical engineering, electrical engineering, civil engineering, etc.), which, in turn, spawned yet more refined sub-specialties, and so on. Each domain is characterized by its own refinements of the

concepts that it inherits from its more general domains. For ease of communication and reasoning, common concepts are given agreed upon domain-specific names that constitute the unique technical vernacular of that domain; i.e., its domain-specific language.

Clearly, when writing software for applications in a given domain, it would be advantageous to have a direct way of expressing these concepts. For instance, the Fortran language provided a more-or-less direct way of specifying mathematical formulas in programs, as opposed to expressing them indirectly through assembly- or machine-language program fragments.

3.1 Expressing Domain Concepts in Software

In general, in software engineering there are two basic strategies for capturing domain-specific concepts. The first approach is through *domain-specific libraries* or *frameworks*. These usually take the form of collections of subroutines, macros, or, in object-oriented languages, classes and methods, which capture the domain concept semantics. The alternative strategy is through *computer language definition*, that is, by defining languages that directly capture domain concepts as first-class language constructs. For example, the Cobol language, which targets data processing applications where it is often necessary to sort data in some fashion, provides a *sort* instruction. This not only saves programmers the effort of having to write their own sort routines but also avoids the possibility of incorrectly programmed implementations of sorting.

Given that the language-based approach involves the extra effort of defining a computer language and at least producing and validating a compiler for it (all of which are technically challenging tasks requiring highly-specialized expertise), what advantages does it have over the library approach? After all, there does not seem to be much of a difference in terms of effort or clarity in calling a procedure as opposed to invoking a language construct. Furthermore, the library approach is more flexible, since it is typically much easier and less risky to modify the code in a library than it is to change the language definition.

However, there are two important advantages of the domain-specific language approach over program libraries. The first of these is the matter of *syntactical form*. Namely, a domain-specific language provides the ability to define a syntax for the language that is best suited to the domain. Consider, for example, the syntax of the Lisp language, which is based on an implementation of the Lambda calculus. It is a syntax that is quite different from the one used in more common procedural languages, but which is very close to the Lambda calculus that inspired it. (Although many software practitioners will claim that syntax is a secondary or even irrelevant concern, this is contradicted by the fact that there are frequent and often heated debates among them on the relative benefits of one syntax over another. *Syntax matters because it contributes to understanding*.) While the linear form of traditional text-based programming languages does not provide for much syntactical differentiation, syntax becomes a much more relevant factor in modern model-based computer languages, which often resort to highly varied graphical representations.

A second and perhaps more important advantage of the language-based approach to domain specialization is that, under normal circumstances, the definition of a language is independent of any specific applications[1]. In effect, the definition of a computer language acts as a kind of standard, whether or it is formal (*de iure*) or an informal (*de facto*). Like all standards, it represents a point of agreement between multiple parties. The language definition can be used by compiler specialists to write compilers, educators to develop training courses, as well as program analysis experts to define methods and build tools that can verify or predict certain important properties of applications written using that language—independently of any specific applications.

Of course, methods and tools can also be constructed to deal with program libraries, but, as noted earlier, libraries are generally much less stable than language definitions. Any modifications to a library can render useless any tools and facilities constructed for the previous unmodified version.

We can conclude from the above general discussion that the choice between a library-based approach or a language-based approach depends on a number of factors and that the choice does not automatically come down in favor of one or the other. Nevertheless, in the remainder of this paper, we will only consider the language-based approach and, more specifically, the issue of *domain-specific modeling languages (DSMLs)* [5][4][6][2].

3.2 General-Purpose versus Domain-Specific Computer Languages

In the domain of *modeling* language design, there has been some theological controversy recently about the relative merits of DSMLs compared to *general-purpose modeling languages (GPMLs)* such as UML [12]. It has been argued that the general nature of GPMLs forces them to be large and unwieldy, making them difficult to learn and difficult to use. Worse yet, because their concepts are general, they do not provide sufficient expressive power to allow concise and precise specification of the kinds of subtleties that characterize many complex application domains. In contrast, DSMLs can have custom-designed concepts, which can be defined as accurately as desired.

These are valid and rather compelling arguments that clearly favor DSMLs over GPMLs. However, when designing a computer modeling language it is useful to consider a number of pragmatic issues that extend beyond purely technical factors.

(At this point, it is useful to keep in mind that the difference between GPMLs and DSMLs is simply a matter of degree of specialization and not a question of some fundamental qualitative difference. What may be characterized as a domain-specific concept from one perspective may be seen as insufficiently specific in the context of a particular application or project. So, even though we opt for a domain-specific language, we may still not get the expressiveness we desire and may still have to resort to some additional form of specialization.)

When considering the question of language specialization, it is instructive to review the history of the development of high-level programming languages. With the invention and subsequent success of Fortran, high-level languages became the standard for

[1] There is some tension between being domain specific and application independent: the more application independent a language is, the less domain-specific it becomes and vice versa. The most domain-specific language is a language that is defined for just a single application.

computer programming, quickly displacing assembly-level programming. The benefits of domain-specific languages were recognized early on, and, in the sixties and seventies of the past century we saw a profusion of hundreds of domain-specific programming languages. There were business data processing languages, report writing languages, artificial intelligence languages, real-time languages, telecom languages, simulation languages, and so on. Yet, although many of these specialized languages are still around and are still being used, the vast majority of current industrial software development is being done using dominant general-purpose programming languages, including the original stalwarts such as Fortran and Cobol, as well as the more recent general-purpose additions (C, C++, C#, Java). To support domain-specific concepts, developers are primarily relying on domain-specific libraries and frameworks. The trend towards highly-specialized domain-specific programming languages seems to be diminishing. Why?

3.3 Necessary Criteria for Successful Computer Languages

Frustrating as it may be to technically-oriented individuals, technical excellence is only one of the factors that contributes to the acceptance or rejection of a computer language. The following are some of the essential criteria that must be met for a computer language to be successful[2]:

- Obviously, first and foremost, the language has to be *technically sound*—it should not have any major design flaws and constraints.
- It should be *expressive*, which means that it should provide constructs that allow succinct and precise specification of concepts from the application domain.
- A language must be *understandable*, which is to say that it should not be so complex to pose a major learning and tooling hurdle.
- For the same reason as above, a new language should have a look and feel that is *familiar* to its target user community. (Languages that introduce new and unusual syntactical forms are often rejected outright by many practitioners.)
- It should be *efficient*, that is, it should be possible to produce with it programs that are sufficiently responsive and require reasonable resources to execute.
- Last, but certainly not least, a language must have an *adequate support structure*.

The support structure of a language consists of a number of different elements. Perhaps the most important of these is the availability of adequate and relatively inexpensive tool support. Specifically, this includes industrial-strength compilers, editors, debuggers, build tools, version control tools, analysis tools, and so on. Without these, a language is very not very likely to gain a significant foothold among practitioners, no matter how technically advanced it might be. Another key element of the support structure is the ready availability of teaching materials and training courses. Development and maintenance of professional-quality tools, teaching materials, and training courses require highly-specialized skills and significant resources that are typically only available from

[2] "Successful" in the sense that it is being used for industrial software development and has a significant and at least a non-diminishing user base. There are, of course, other valid definitions of success, but those are out of scope of this overview.

specialized commercial vendors or from large-scale open source projects. It is difficult to secure these for smaller niche languages. Consequently, the only languages that truly have adequate support structure are general-purpose languages, ones with significant populations of users.

Based on the above, it seems that the right question to debate is not whether a language is "domain-specific" or "general-purpose", which, as we have pointed out, are rather vague terms, but how well it meets the above criteria.

3.4 Approaches to DSML Design

To date, three primary methods for defining DSMLs have emerged:

1. Refinement *of an existing modeling language* by specializing some of its general constructs to represent domain-specific concepts.
2. Extension *of an existing modeling language* by supplementing it with fresh domain-specific concepts with new constructs that are derived from the existing language concepts.
3. *Definition of a* new *modeling language* from scratch.

Without doubt, the last of these has the potential for the most direct and succinct expression of domain-specific concepts. However, it suffers from the serious drawbacks discussed in section 3.3, particularly in terms of lack of an adequate support structure. Furthermore, it is generally more difficult and expensive to develop tools for modeling languages than for programming languages, due to the usually more sophisticated semantics behind many modeling language constructs. For example, a tool that compares two different versions of a state machine model must “understand” the semantics of state machines. In contrast, because programming languages are mostly textual, the differences are usually expressed in terms of lines of text that have been added or changed, without any concern for semantic constructs, such as states or transitions. The semantic interpretation of such differences is left to the programmer. Unfortunately, this kind of semantics-free differencing is not practical for graphical languages. Similar issues exist with other kinds of tools, especially those intended for model analysis.

In essence, the same set of problems is encountered in the second method albeit in a somewhat milder form because some degree of support structure and expertise reuse can be expected.

Consequently, the refinement-based approach seems to be the most practical and most cost-effective solution to DSML design in many situations. If properly designed, an extension-based DSML allows reuse of the tooling support structure of the base language, access to a broader base of trained experts, and usually requires less specialized training. On the other hand, its principal disadvantage is that the expressive power of the DSML may be diminished due to the semantic and syntactic limitations of the base language. Therefore, it would be wrong to conclude that this approach is optimal in all situations. However, if the base language is relatively general, the likelihood that it will be limiting is actually much less, because it has general constructs that leave more opportunity for refinement.

This is the rationale behind the so-called *profile* mechanism provided in UML 2. Unfortunately, this mechanism was designed in piecemeal fashion, starting with a

simple (simple-minded?) understanding of the problem and a simple solution, evolving gradually into something more sophisticated as more experience and understanding accrued. Furthermore, the semi-formal nature of the UML 2 language and its metamodel compounded the difficulties. The refinement approach is based on the notion of semantic containment, which is very difficult to verify unless there is a formal foundation to support it. More specifically, it is not easy to ascertain whether a particular UML profile is a proper refinement or an extension of UML proper.

Nevertheless, the idea of a profile as a semantically contained refinement of some more general base language is a useful one due to its obvious advantages (reuse of tools, etc.). We will examine the UML 2 implementation of profiles in section 5.

3.5 The Fragmentation Problem

DSMLs are often used to specify different viewpoints of a complex system. Each viewpoint deals with one set of concerns and, therefore, is an abstraction of the underlying full system that emphasizes features related to those concerns while ignoring or hiding from view those that are not. For sufficiently complex systems, multiple viewpoints are necessary, which means that it is highly likely that some features will be represented in more than one viewpoint. This means that we may have multiple descriptions of the same feature, expressed in multiple models, each of which could be specified using a different DSML. The so-called *fragmentation problem* is the problem of ensuring that the different representations of a given system feature, specified using different modeling languages, are mutually consistent. This problem is greatly compounded if the DSMLs used are semantically unrelated (e.g., use different models of computation).

A pragmatic way of coping with this problem is to have an underlying merged representation of the system being modeled. This representation is constructed by merging the information provided by individual DSML models. The different domain-specific views can then be viewed as projections of the underlying merged model. When a change is made to one of the projections, it is translated into a modification of the underlying merged model, where any inconsistencies can be detected and flagged. The merged model, being a model, is also expressed using some modeling language, possibly a more general language (since it has to somehow accommodate the information defined in all the different domain-specific models). Also required are two-way mappings between each of the DSMLs and the merged model language.

It seems self evident that, if the various DSMLs share a common semantic base, then these mappings should be simpler to define than if the languages are independent. Once again, it seems that a profile-based strategy for DSMLs has an important advantage here, provided that all the DSMLs are refinements of the same base language.

3.6 Refine, Extend, or Define?

Although we mentioned some advantages of the refinement (i.e., profile) approach, the question of which approach is best has no single easy answer. It depends very much on the problem at hand. The refinement approach should be considered only if there is a

significant semantic similarity between the concepts of the desired DSML and the concepts of the chosen base language. Furthermore, there should be no semantic conflicts, such as contradictory constraints, between the base language and the desired DSML. If these conditions are not satisfied, then refinement is probably not a good choice and some other approach may be more suitable.

There are other pragmatic considerations to take into account when choosing an approach to designing a DSML. The following seem to be some of the most important:

- When designing a DSML from scratch, it is extremely useful to have someone with modeling language design experience available for consultation, since this still far from being an exact science. There are numerous pitfalls in designing modeling languages (we will discuss some of them later) that can cause much grief in both the definition and the use of a language. The refinement and extension approaches hold the advantage here since they already embody such experience.
- It is also highly recommended to have direct domain expertise at your disposal during the process of language definition. A well-designed DSML draws a balance between the inevitable technical constraints imposed by computer technology and the needs of domain experts. Language design experts tend to favor the former and, because they lack domain insight, they often have a distorted view of the domain requirements.
- An assessment must be made regarding the anticipated costs of establishing and maintaining an adequate language support structure (compilers, debuggers, miscellaneous utilities, libraries, training materials and expenses). These can be quite substantial if a completely new language is being contemplated.
- Another important consideration is the ability of the language to interwork with other languages (e.g., legacy code). By their very nature, DSMLs focus only on specific aspects of a complex system, and, consequently, specifications written in these languages will likely need to relate in some way to specifications written using other DSMLs. To this end, some authors suggest that a DSML definition should include explicit specifications of its required and provided interfaces to other languages [5]. For example, a language might expose certain of its concepts to be referenced by other languages. Similarly, it may specify external foreign concepts that it needs to reference.
- Related to the above, an important issue to consider is the ease with which the support structure of a DSML can be seamlessly integrated into an existing development environment.

4 Metamodeling

Metamodeling is a technique for defining modeling languages. It consists of defining a special model, the *metamodel*, using a modeling language designed specifically for metamodeling, which defines the language concepts and their relationships—its abstract syntax. Note that the metamodeling DSML is often supplemented by other languages, such as a language for formally capturing constraints (e.g., OMG's OCL).

4.1 Context-Free Grammars versus Metamodels

The technique of using models to capture the abstract syntax of a modeling language is a departure from the traditional programming language design practice of using *context-free grammars* for that purpose. Context-free grammars are usually expressed through some variant of the Backus-Naur Form (BNF)—a text-based syntax. In contrast, most metamodeling languages use so-called *graph grammars*. It seems that metamodeling languages, such as the OMG's MOF [10], often provide more concise and more readily discernible abstract syntax specifications than context-free grammars. This is particularly true where complex relationships exist between different language concepts, a very common case. Such relationships can be represented directly through graphs and are easier to comprehend, particularly if a graphical concrete syntax is used. Context-free grammars, on the other hand, are better suited to describing simple tree structures. This complicates the specification of general graphs, since it requires the use of references to represent complex graph structures.

Moreover, metamodeling languages allow language concepts to include attributes, allowing for a more concise representation. This also enables the use of inheritance mechanisms which can further simplify language definition.

Finally, with metamodeling it is possible to incorporate various constraints domain-specific constraints directly into the metamodel. With context-free grammars, these constraints need to be kept separately.

4.2 Metalevels

We have already noted that the language used to define a metamodel, is itself a modeling language, which, of course, can have its own metamodel. Naturally, the latter is also expressed using some modeling language, and so on. Clearly, this could lead us to an infinite regression of metalevels. The usual method of circumventing this recursion is to make one level self-defining. That is, at some level the metamodeling language at that level is used to define itself. A convenient terminology was introduced to help us differentiate the different levels in discussions. The hierarchy starts off with the actual system that needs to be modeled. It is referred to as *meta-level zero* (or, simply, *M0*). For example, this might be some planned or existing software program that we would like to model. Models of M0 entities occur at meta-level one (*M1*, such as, for example, a UML model of some software. The (meta)model that describes the M1 modeling language is at level M2. In case of the UML standard, this language is the *MetaObject Facility (MOF)* language. Most such hierarchies end with level M3, with the definition of the meta-metamodeling language, which is defined in terms of itself. In the OMG language hierarchy, this is yet again the MOF language.

4.3 The OMG's MOF

The MOF is a typical example of a present-day metamodeling language. Another, quite popular metmodeling language is the Eclipse Modeling Framework (EMF) [1], which is much simpler than the MOF, but less expressive. Most metamodeling languages can be described as simplified versions of class-association modeling found in UML. In fact, both UML 2 and the MOF share a common library, called the UML 2 Infrastructure, which captures this commonality. However, despite this syntactical

similarity it should be kept in mind that the domain of MOF—the modeling of modeling languages—and the domain of UML—modeling of software-based systems—are quite distinct from each other. Thus, since the MOF is mostly used to describe static abstract syntax structures, unlike UML, it has practically no facilities for specifying behavior.

The core constructs of the MOF are illustrated in the example in Fig. 2 (the names of the constructs are indicated by the text inside the dashed line callout boxes – note that these are not actually part of the MOF language but are provided for convenience). These constructs should be familiar to anyone familiar with UML class diagrams, and will be described only briefly here.

This particular example shows a fragment of the abstract syntax definition for a simple DSML for modeling applications from the automotive domain. Language concepts are specified by *classes* (e.g., Vehicle, Automobile, Person). Note that abstract concepts, that is concepts which are not directly usable by modelers (and, hence, without a concrete syntax), are indicated by the fact that their names are italicized (e.g., Vehicle). Language concepts can be refined by providing them with typed attributes, such as the "id" attribute of Vehicle (which might be used to specify some type of identification string).

Language concepts can be related to each other via *generalization* relationships. In this case, we see that an Automobile is a special kind of Vehicle which means that it inherits all the properties of Vehicle, such as "id", but which also adds further Automobile-specific attributes, such as "make" and "power".

Concepts can also be related to each other via *associations*. Associations signify that the use of one concept in a model may require the presence of a related concept at the opposite end of the association. For instance, we can see that if the model includes an instance of the Automobile construct, it might be accompanied by a "driver", which is an instance of the Person construct. In fact, the notation "0..*" specifies that the number of "driver" instances is open ended, starting with zero (i.e., no "drivers").

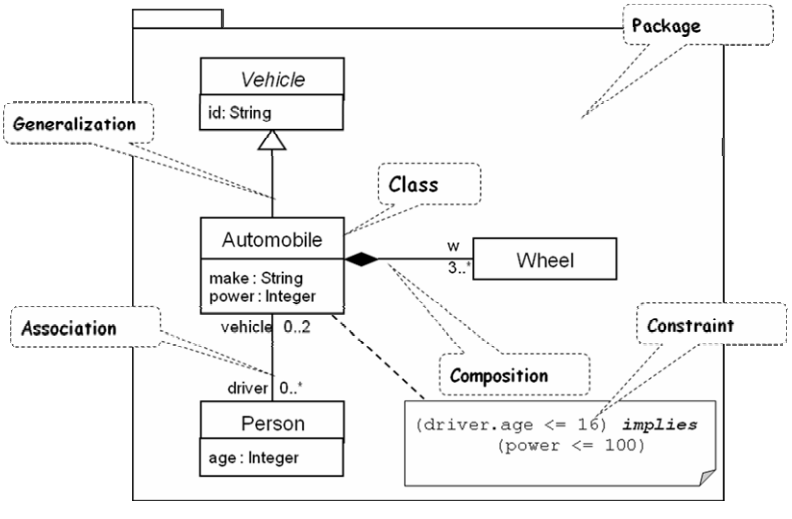


Fig. 2. Basic MOF modeling concepts

A special kind of association is known as a *composition*. A composition means that, if an instance of the owning concept (indicated by the class adjacent to the end with the filled diamond graphical element) is removed (e.g., Automobile), then the corresponding instance(s) at the opposite end will also be removed (e.g., instances of Wheel).

As mentioned earlier, abstract syntax specifications are often accompanied by constraints, which provide some additional rules that a well formed model must respect. In the example, the rule is specified using the OCL language and indicates that if the "power" of the Automobile is equal to or greater than 100, its drivers must be at least 16 years old.

MOF *packages* are not used to capture any language concept. Instead, they are used to group elements of the abstract syntax definition into convenient modular units, either as units of reuse or as a way of partitioning a complex specification. MOF packages are namespaces whose elements can be designated as public, protected, or private. A public element can be referenced from outside by elements in other packages. This can be achieved by using the *package import* mechanism of MOF. That is, when a package P1 imports package P2, elements in package P1 can directly reference the public elements of package P2 as if they were defined within package P1.

MOF provides another useful mechanism related to packages: *package merge*. The basic idea of package merge is to enable incremental definition of language concepts. It allows a base concept definition to be selectively extended with incremental definition fragments. For instance, consider the metamodel in Fig. 3a, where the ResPackage contains a definition of the Element concept, which is defined as a subclass of the Object concept. When the contents of the merge increment defined within the IncrementalPackage are merged into the ResPackage, the result is shown in Fig. 3b.

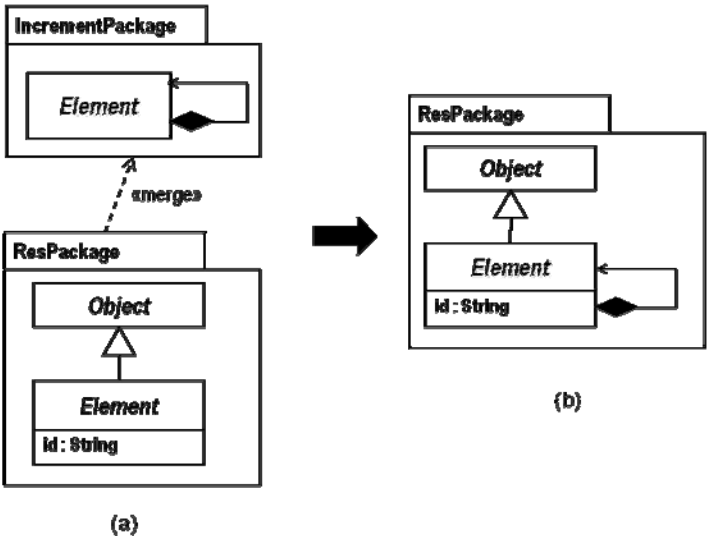


Fig. 3. Package merge

The package merge operation works by finding matching elements with the same name in the two packages and merging them into a single combined element in the merging package. It is a mechanism that is quite similar to generalization in effect since it merges two sets of features into one. However, in contrast to generalization, the result is a single concept rather than two concepts with different names.

Package merge is particularly useful when defining languages that have multiple levels of definition. For example, at one level of definition, the Element concept from Fig. 3a might be defined simply by the contents of IncrementPackage. This may be sufficient for some users of the language, who would prefer not to be bothered with the more sophisticated form in Fig. 3b. However, for those users who need the more complex definition of the concept, the language can be easily extended by merging the desired increments. Note, furthermore, that valid models based on the simpler definition will still be valid in the extended language definition (but not vice versa). This allows the possibility of smooth conversion of models as the definition of the language evolves.

4.4 Mixin-Like Concept Definition

In traditional object-oriented terminology, a "mixin" is a small feature that is defined independently and which, like trace ingredients in a food recipe, can be combined with another more substantial definition, to give the overall result an additional "flavor" (i.e., capability). Mixins are similar in intent and method to aspects as encountered in aspect-oriented programming. For instance, it may be useful to define Redness as a distinct concept that captures the fact that something is of a red hue. This can then be composed with other concepts, such as Hair, Automobile, or Sky, to produce the notions of RedHair, RedAutomobile, and RedSky respectively. The advantage of doing it in this manner is that Redness is defined in just one place, independently of other definitions. This allows cleaner and more precise definition of concepts and also enables independent modification of that concept should the need arise.

The idea of mixin-like fragments in combination with generalization as a way of merging the fragments has emerged as a useful metamodeling style. It works as follows: a core of fine-grained and independent language features is defined, usually as abstract concepts. They are then composed into various useful feature combinations

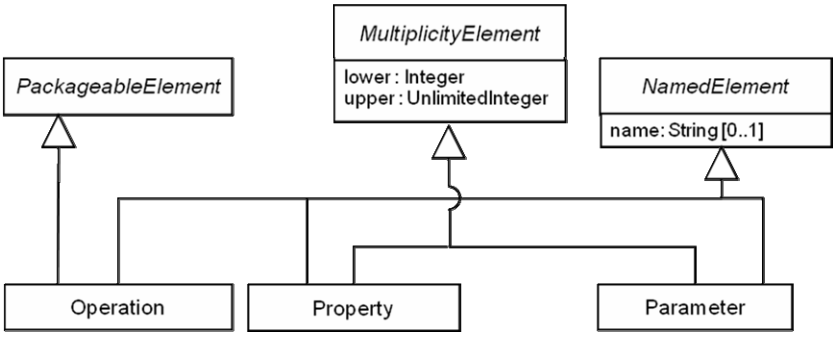


Fig. 4. Mixin-like concept definition

using multiple generalizations[3]. Consider, for example, the language definition fragment shown in Fig. 4. In this case, the mixin elements are shown at the top; PackageableElement captures the notion of a language construct that can be contained in a package, MultiplicityElement represents an element that is somehow bounded within an integer range, and NamedElement is an abstraction of a concept that may have a name. Each of them is defined independently of the others. However, they are combined in different ways to produce different language constructs. Operation is a combination of PackageableElement and NamedElement. Parameter combines NamedElement and MultiplicityElement as does Property.

While this approach is quite elegant and has the obvious advantages, it should be used with care. Since it uses multiple generations to combine mixins, it has the usual problems of multiple inheritance, such as the potential name clashes and diamond inheritance. Care must be taken that the mixins are indeed independent concepts, or they may interfere with each other. Finally, this approach is also particularly susceptible to another major problem, overgeneralization, which is described in the following section.

4.5 The Overgeneralization Trap

Overgeneralization occurs in deep generalization hierarchies due to unanticipated conflicts between multiple conflicting generalizations. As an example, take the fragment of the UML 2 metamodel shown in Fig. 5. In this case, the top abstraction is the notion of an Element, which captures certain characteristics that apply to all language concepts. One of these is the notion of ownership shown by the composition from Element to itself. In essence, this is saying that an instance of Element may also own other Elements. In fact, this is where the core concept of ownership is defined in UML (this concept is specialized in many different places in the metamodel, not shown in this diagram). Two immediate refinements of Element are the abstract concepts of Classifier (i.e., something that can be classified into generalization hierarchies) and Relationship. The problem of overgeneralization can be seen even at this level: note that, since no additional constraints are imposed on the Relationship and Classifier concepts, it is possible for a Relationship to own a Classifier and vice versa[4]. The problem is even less visible the deeper one goes in the generalization hierarchy. For instance, if Dependency is a kind of Relationship (several levels below) and UseCase is a kind of Classifier (also several levels below), then a Dependency can own a UseCase—which is definitely not the intent of the language designers.

The conflict comes when the general notion of ownership, as defined for Element, is combined with the notion of specializations of Element. For each specialization of Element, there needs to be constraints that limit the ownership association to only the appropriate types. Furthermore, this has to be repeated transitively for every subclass, which is not only tedious, but also makes the metamodel difficult to modify.

---

[3] It might appear at first that this can be achieved by package merge as well. However, package merge uses name matching whereas the mixins all have different names and may have to be combined in many different ways.

[4] A well-documented manifestation of this problem encountered in OO programming languages is known as “co-variance”.

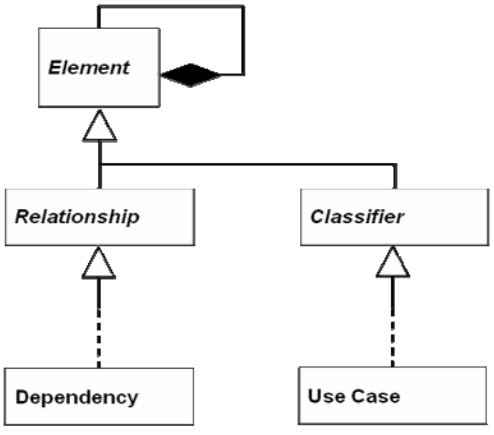


Fig. 5. The problem of overgeneralization

The problem can potentially occur with any feature (attribute, association end, or operation) defined for abstract concepts. In principle, each time a subclass is defined, all of its features need to be examine for such conflicts and appropriate constraints formulated.

4.6 Some Lessons Learned and Guidelines Related to Language Design

The author participated in the definition and standardization of both the MOF and UML languages. The following are some lessons learned from this experience. Modeling language designers may choose to use them as guidelines or as warnings of potential pitfalls that might await them:

1. *Always start design with a semantics model.* Before getting into the business of designing the abstract and concrete syntaxes, it is necessary to have a sufficient understanding of the domain. In particular, for computer language designers, it is useful to think early about a suitable model of computation that is a close semantic match to the domain semantics (we will say more on this in section 7.1). This will not only facilitate and ease language design (because a model of computation typically implies numerous consistent design decisions that can be reused), but may also provide support for different kinds of formal computer-based analyses of models. For instance, if the model of computation is based on Petri nets or on some kind of finite state machine formalism, then it becomes possible to take advantage of many formal analyses methods defined for these formalisms.
2. *Modularize and layer the language*, particularly if the language is going to support multiple viewpoints for different sub-domains. Note that a language modularized in this way can allow the use different syntaxes (abstract and concrete), for each sub-domain. To avoid the fragmentation problem, however, it is generally useful to have a common foundation that supports all the different modules and provides a means for ensuring consistency of models.

3. *Provide for language extensibility.* No matter how confident you are about your understanding of the domain and its users, chances are that you will need to specialize the language in the future. Some kind of relatively lightweight extensibility mechanism is useful in such cases, so that the impact of the extension on existing tools and other language support structure can be minimized.
4. *Allow for incorporating "foreign" language fragments in models.* In far too many cases, a language is rejected because some relatively small but crucial part of a system cannot be properly or efficiently specified in that language. Rather than throw out the baby with the bathwater, it may be useful to allow for incorporating fragments of other languages in a model. This is the modern equivalent to the old facility provided in many earlier programming languages of incorporating assembler code fragments into a high-level language program. (Note that sometimes, this problem can be resolved by providing custom code generation capabilities for such fragments, but this is still not an exact science. Until this issue is resolved, it is a good idea to allow some kind of language infiltration.)
5. *Beware of overgeneralization* (see section 4.5).
6. *Make a clear distinction between language constructs that represent design-time concepts from those that represent run-time concepts.* Experience with UML in particular has shown that, unless these concepts are clearly delineated, modelers will confuse them and use design-time concepts to represent run-time ideas. A UML package, for example, is strictly a design-time concept, yet it is often used inappropriately to model some run-time concept, such as a layer.

5 Profile-Based DSML Definition

The *profile mechanism* of the Unified Modeling Language (UML) was designed to support the refinement approach to DSML design. It is restricted to DSMLs that fall within the syntactic and semantic envelope defined by standard UML. This is because, by definition, a UML profile cannot violate any of the abstract syntax rules and semantics of standard UML.

From its inception, UML was designed to be customizable, to be a "family of languages". Consequently, its definition includes numerous *semantic variation points* and it provides mechanisms for its own refinement (the profile mechanism). Semantic variation points are areas in which the UML specification supports multiple possible interpretations. These may be explicit in the form of multiple pre-defined choices, or implicit, by not leaving undefined certain semantics. Semantic variations can be reduced or eliminated by adding constraints or using other refinement mechanisms such as stereotypes. For example, standard unrefined UML supports both single and multiple generalization (inheritance). However, this can be limited to just single inheritance simply by defining one additional constraint that limits the number of ancestors of a class to no more than one.

The basic refinement mechanisms of the initial versions of UML, *stereotypes* and *tagged values*, were relatively lightweight and, unfortunately, not very precisely defined. They permitted attaching domain-specific semantics to selected elements of UML models. For instance, a particular class in a UML model could be selected to represent a mutual-exclusion semaphore device by tagging that class with a custom-built "semaphore"

stereotype. By attaching this stereotype to a given model element, that element automatically acquires the semantics associated with the "semaphore" stereotype, in addition to its standard UML class semantics. Conversely, removing the stereotype from the model element results in the removal of its semaphore aspect, while retaining its original base semantics.

A *stereotype definition* consists of a user-defined stereotype name, a specification of the base UML concept (e.g., Class) for the stereotype, optional constraints that specified how the base concept was specialized (e.g., a Class that can have at most one parent), and a specification of the semantics that the stereotype adds to the base concept semantics. The latter is typically specified using informal natural language, but, of course, formal specifications are also possible.

A UML tool supporting only standard UML will treat an element with an attached stereotype will simply treat the stereotype like an attached comment that it can ignore. However, a more specialized tool or a custom version of a standard UML tool that is sensitive to the semantics of that stereotype, will detect and inspect the stereotype and interpret it appropriately. For example, a concurrency analysis tool might be able to detect potential concurrency conflicts in a model by analyzing the accesses to those model elements tagged with semaphore stereotypes.

Since stereotypes capture domain-specific concepts, they are typically used in conjunction with other stereotypes from the same domain. This eventually led to the concept of a *profile*, a specially designated UML package that contained a collection of related stereotypes.

5.1 Innovations to Profiles Introduced in UML 2

Many of the shortcomings of the initial profiling mechanism, stemming mostly from its imprecise definition, were eliminated in UML 2. Unfortunately, these improvements have received very little coverage in popular UML textbooks and are still relatively unknown and underutilized. The following are the most important of these innovations:

- An expanded and more precise definition of profiles and stereotypes was provided. Thus, in UML 2, a stereotype is semantically very close to the concept of a metaclass (i.e., a language concept in the abstract syntax).
- Formal rules for writing OCL constraints attached to stereotypes were introduced. Thus, even a standard UML tool could detect if a constraint within a profile violated some standard UML constraint.
- A new and more scalable notation for stereotypes and their attributes was added.
- The semantics of applying (and un-applying) profiles to UML models were both expanded and clarified.
- The rules for the serialized (XMI) representation of profiles and their contents were defined.
- It became possible for a profile definition to be based on just a subset of the UML metamodel (as opposed to the full metamodel), resulting in potentially very compact and simple DSML specifications.

- The ability to create associations between stereotypes and other metamodel elements was added. This allowed the creation of new relationships between previously independent UML concepts.
- A default "batch" stereotyping mechanism was introduced to simplify the stereotyping of individual model elements.
- The profile mechanism was generalized beyond the UML context so that it could be used with any MOF-based modeling language.

In the remainder of this section, we provide a brief description of the principal features of the UML 2 profile mechanisms.

5.2 Profiles

There are two significantly different ways in which UML profiles can be used:

A profile can be created *to define a DSML*. This language can then be used to construct domain-specific models using concepts from that DSML. An example of such a profile might be a language for modeling real-time applications. This language might provide domain-specific concepts such as Priority, Task, Deadline, etc. in place of the corresponding general-purpose UML concepts such as Class, Behavior, etc.

Alternatively, a profile can also be created to define a *domain-specific viewpoint*. Such a viewpoint, when applied to a standard UML model, re-casts selected elements of that model in a domain-specific form. The result is a domain-specific interpretation of the original model. Furthermore, the profile may also add supplementary information relevant only to that viewpoint. For instance, it may be useful to analyze the performance characteristics of a particular design expressed as a UML model. One common technique for analyzing performance characteristics of software systems is based on queueing theory, which views a system as a dynamically balanced network of interacting clients and servers. Using a performance UML profile, it is possible to identify individual model elements in the base model as playing the roles of clients or servers by attaching appropriate stereotypes to them, including element-specific performance metrics, and then to compute the overall performance characteristics of the proposed design.

The ability to dynamically apply and un-apply a UML profile without affecting the underlying model is crucial to this type of profile usage, because it allows the same model to be viewed from any number of different viewpoints (e.g., performance, security, availability, timing).

Typically, a model to which a viewpoint profile has been applied is transferred to a specialized analysis tool that can then transforms it into an appropriate analysis-specific model based. This model can then be analyzed by the domain-specific tool and the results fed back into the original UML model. By this technique, modelers can take advantage of many useful types of analyses without having to be experts in those techniques. This can be of great utility since many of these analyses are quite complex and require levels of expertise and skill that are both scarce and expensive.

5.3 Stereotypes

As noted previously, stereotyping of model elements is a convenient way of identifying elements in a UML model that have additional non-standard semantics; i.e., semantics

that extend beyond the UML standard itself. Stereotype definitions are often supplemented with constraints (typically written in OCL). These are used to capture domain-specific constraints that apply only to the stereotype but not to its base UML concept.

A UML stereotype is almost like a subclass that specializes a standard (*base*) UML modeling concept (specified by a metaclass). However, there are some important differences. One of the primary reasons for this is the need to support viewpoint type profiles described above, which require the ability to dynamically apply and un-apply profiles.

Namely, when a stereotype is applied to a model element, it is in the form of a special attachment to that model element. This attachment contains the information about the applied stereotype and any values associated with its attributes. When the corresponding profile is removed (i.e., when the viewpoint is no longer needed), the attached stereotypes are simply removed without affecting the original model element in any way.

Another important reason for the difference between stereotypes and regular metaclasses, is that a stereotype can specialize more than one base metaclass. However, the semantics of this are *not* the conjunctive semantics of multiple generalization. Instead, the semantics are disjunctive, which means that the stereotype can be applied to model elements that are instances of any of its base metaclasses (or any of their subclasses). This feature allows a given domain-specific concept to be realized by more than one base UML concept. A typical case where this is useful is when we need to apply a stereotype to either a type (e.g., Class) or to instances of a type (model elements typed by InstanceSpecification).

Modelers can apply stereotypes selectively to model elements of their corresponding base class. For instance, if «clock» is a stereotype of the Class metaclass, then it is not mandatory for all model elements that are instances of Class to be stereotyped by this stereotype. However, in some cases, it may actually be required that a stereotype must be applied to all instances of the base metaclass. For instance, if a model represents a C++ program, then all classes in the model should have the same C++ stereotype applied, since C++ only recognizes one type of class. For those situations, the profile designer has the choice to declare the stereotype to be "required", which means that all instances of the base class and its subclasses *must* have the stereotype applied to them whenever the corresponding profile is applied.

5.4 Model Libraries

A model library is a stereotyped package[5] that contains useful model fragments intended to be reused by other models, most notably profiles – although they can be reused by any model. If the library is defined in the context of a profile (package), then it is part of the profile definition. This allows complex domain-specific concepts to be captured using the full power of standard UML modeling constructs, unhampered by the limitations of stereotype modeling. A common application of such libraries is to use them to type stereotype attributes.

However, it is important to note that, even when a library is part of a profile, the elements it describes are not metamodel (M2) elements but M1 model elements. This

[5] Model libraries are packages identified by the «modelLibrary» system-defined stereotype.

means that, unlike stereotypes, they do not have any special semantics outside those provided by UML (unless, of course, they are themselves stereotyped). Still, they can be used to capture domain-specific semantics by association, so to speak, due to the fact that they are defined in the context of a particular profile.

For example, a model library for robotics, might introduce classes such as Robot, Manipulator, VisionSystem, etc. Outside the context of a profile, these are just classes with nothing to differentiate them from any other UML model elements. However, when they are used within or part of a profile, they implicitly acquire domain-specific semantics by association. In such situations, a tool that is sensitized to the encompassing profile can interpret these elements in a domain-specific way.

As a special degenerate case, a profile may be defined without any stereotypes, containing (or importing) nothing but model libraries. The drawback of this, however, is that such a profile cannot take advantage of some of the features of profiles, notably the ability to be dynamically applied or unapplied.

6 A Systematic Method for Defining UML Profiles

There has been little material published to guide designers of UML profiles. The consequence is that there are very many UML profiles that are either technically invalid because they contravene standard UML in some way (and, thus, cannot be properly supported by standard UML tools) or they are of poor quality. In this section, we describe a method for defining profiles that will avoid some of the most common pitfalls.

In this approach, the definition of a UML profile involves the construction of two distinct but closely related artifacts: a *domain model* (or metamodel) and the *profile* itself. The process commences with the initial definition of the domain model, which is then translated into the profile. However, depending on the complexity of the DSML and how well it conforms to UML, it may be necessary to iterate between these two artifacts. In addition to domain expertise, a close familiarity with the UML metamodel is also vital when defining a profile.

6.1 The Domain Metamodel

The primary purpose of a domain model is to specify what needs to be represented in the DSML and how. Experience with defining profiles has indicated that it is best if the initial domain model is defined strictly on basis of the needs of the domain, without *any* consideration of the UML metamodel (to which it will be mapped subsequently). This achieves a useful separation of concerns and ensures that the initial DSML design is not corrupted by contingencies of the profile mapping.

Unfortunately, far too many profiles start with the UML metamodel trying to fit the various domain concepts one-by-one within the framework that it provides. This conflates domain modeling issues and profile mapping issues and typically leads to loss of focus on domain concerns. The usual result is a DSML that is well aligned with the UML metamodel but which are suboptimal for domain modeling.

An “ideal” DSML metamodel, on the other hand, is an unpolluted specification of what the corresponding profile should provide. In practice, it may not always be possible

to map this model precisely to the UML metamodel since the latter may have some constraints, metaclass attributes, or relationships, which are in conflict with the domain model. When that happens, it may be necessary to adjust the domain model with some loss of expressive power as a result. Naturally, if the profile strays too far from the ideal, then it may be the case that the profile-based approach is inappropriate or that UML is not the right base language for that particular DSML. (Note that, in such cases, the work invested in developing the domain model can be fully reused for a non-profile DSML.)

A domain model is metamodel of the DSML that should all the key elements of a modeling language described in section 2.

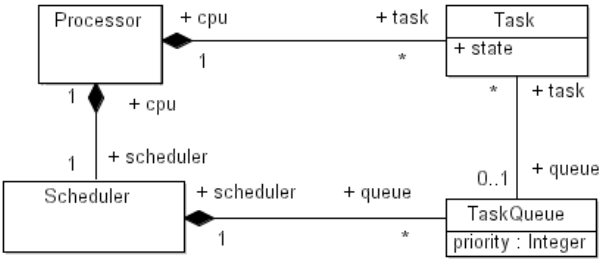


Fig. 6. A partial domain model of a DSML

The abstract syntax of a DSML can be expressed using the OMG MOF (Meta-Object Facility) language. In principle, other metamodeling languages can be used for this purpose, but MOF has the advantage that its models are easily translated into UML profiles. This is because UML metamodel is defined using MOF.

The basic language constructs are typically captured using MOF classes and attributes, while the relationships between them are represented either by associations or through class attributes. As much as possible, profile constraints should be written using OCL, since that language was specifically designed to be used with MOF and because it is supported by many UML tools.

For example, the relationship between processors, tasks, schedulers in an operating system DSML example would be captured using the MOF model fragment depicted in Fig. 6.

6.2 Mapping the Domain Model to a Profile

Once the domain model is completed, the process of mapping it to the UML metamodel can commence. This is done by going step-by-step through the full set of domain concepts (specified as classes in the domain model) and identifying the most suitable UML base concepts for each.

For example, in the operating system DSML shown in Fig. 6, we might conclude that the Processor domain concept is conceptually and semantically similar to the Node concept of UML, while the Task concept is related to the UML Class concept whose “isActive” attribute is set to “true”. Thus, we would define a stereotype called Processor whose metaclass is Node and a stereotype called Task whose metaclass is

Class. We could then define an attribute of the Task stereotype called "state" and also add a constraint that the "isActive" attribute of any Class tagged by this stereotype must have its "isActive" attribute set to "true".

Note that not all domain concepts need to be directly derived from corresponding UML metaclasses; some may be specializations (subclasses) of other "abstract" stereotypes in the profile. For example, a generic "semaphore" domain concept might be defined as a stereotype of the UML Class concept. This stereotype can then be refined further by multiple subclass stereotypes corresponding to different flavors of the generic concept (e.g., queueing semaphores, binary semaphores, etc.).

The richness and diversity of concepts in the UML metamodel provides a very good foundation on which to base concepts for a very large number of DSMLs. However, although it may be easy in most cases to find a corresponding base metaclass for a domain concept, this is not sufficient. Care must be taken to ensure that the selected base metaclass does not have any attributes, associations, or constraints that are contradictory to the semantics of the domain concept. Often, such issues can be resolved by simple constraints. For example, a conflicting attribute or association end from a base metaclass can be eliminated by forcing its multiplicity to be zero using a constraint (provided that it has a lower multiplicity bound of zero).

The following guidelines should be used for mapping domain concepts to UML metamodel elements:

1. *Select a base UML metaclass whose semantics are closest to the semantics of the domain concept*. This is very important since the semantics of UML concepts are often built into UML tools and also because people who know UML will naturally expect a stereotype to inherit the semantics of its base metaclass. After all, the reuse of UML tools and UML expertise are among the primary justifications for the profile approach.
2. Unfortunately, all too often, base metaclasses for stereotypes are chosen on the basis of a purely syntactic match. For instance, contrary to what might be expected, the OMG SysML profile does not use the UML Dependency concept to capture certain kinds of functional relationships that exist in a model. Instead, it uses the Class concept for this purpose because that allows reuse of some notational forms used with UML classes. In general, purely syntactical matches of this type should be avoided since they will lead to confusion and misinterpretation by tools.
3. *Check all the constraints that apply to the selected base metaclass to verify that it has no conflicting constraints.* Note that it may not be sufficient to check just the base metaclass for such constraints but also all of its superclasses, if they exist.
4. *Check to determine if any of the attributes of the selected base metaclass need to be refined.* This is a way of specializing the base concept for domain-specific semantics. For example, in modeling a task in the example operating system DSML, we added a constraint that specified that the "isActive" attribute must be set to "true". Constraints of this type may be used to define domain-specific default values of attributes and also to eliminate attributes that may not be relevant to the domain (by setting their lower multiplicity bounds to zero).
5. *Check to determine if the selected base metaclass has no conflicting associations to other metaclasses.* These would be conceptual links inherited from UML that

contradict domain-specific semantics in some way. Fortunately, many of these can be eliminated by the above technique of setting their lower multiplicity bounds to zero. However, if this is not possible, then this may not be the appropriate metaclass despite its semantic proximity to the domain concept.

7 Semantics

The semantics of a modeling language define what the concepts and rules of the language denote, that is, their meaning. Clearly, for a domain-specific computer language, these are primarily drawn from the problem domain, although, as we shall argue, the technical domain related to adapting these to computing technology must also play a significant role in the semantics. For example, the concepts and the rules have to be formulated as precisely and completely as possible, since they have to be realized by a computer. This usually requires some kind of formal (e.g., mathematical) or highly structured informal specification format. In addition, they must be formulated in a way that is relatively easily mapped to the way that computers operate, not only because we are concerned efficiency and feasibility (which we normally are), but also because that increases the likelihood that the domain concepts will be captured accurately[6].

7.1 Static and Dynamic Semantics and Models of Computation

Static semantics refers to the various rules that specify what constitutes a well-formed (i.e., legal) program. These rules quite often reflect some characteristics of the domain, such as, for instance, rules on compatibility between data of different types. Static semantics are normally captured by the abstract syntax of the language. *Dynamic semantics*, on the other hand, describe the run-time aspects of the language, that is, how programs written in the language execute on a computer[7].

As noted earlier, the inspiration for dynamic semantics normally comes from the problem domain. For a domain-specific language, these semantics should reflect the way that domain practitioners prefer to think about how things happen in their world. However, as the real-world is infinitely more varied and more complex than computing technology, there is a need to map domain semantics to some kind of model of computation. A *model of computation* is a paradigm for how computers

[6] As an example of how technological concerns can impact a domain, consider the difference between the way that numbers are defined in mathematics and the way they are realized with computing technology. In the latter case, we have to deal with issues of finite precision, rounding, overflows, etc., all of which prevent us from a fully accurate rendering of the domain concept. As experience has shown, such factors can indeed have a significant impact on language design.

[7] In rare cases, a domain may not have any noteworthy dynamics. For instance, the MOF language is used for defining static structures and the rules that control them. The models constructed by meta-modeling languages of this type are (typically) not meant to be executed in the traditional interpretation of that term. In such cases, it is sufficient for the language to define just the structural patterns that characterize the domain.

execute programs[8]. Many different models of computation have been defined in the past, including logical, functional, procedural, object-oriented, `quantum, and so on. For our purposes, we will focus only on the following two general categories that are the most commonly used and which are most readily understood by individuals who are not necessarily computer experts:

- *Behavior-dominant* models of computation. This category includes both procedural and flow-based (e.g., functional, data flow) models. They are well suited to domains where algorithms, parallel or serial, are the dominant phenomenon. The base concept here is that of a stateless computational fragment that performs some transformation of its inputs and presents them on its outputs. These fragments can be combined using either control flow relationships (for procedural type models), data flow relationships (for functional models), or some hybrid of the two.
- *Structure-dominant* models represent computation as a network of collaborating computational entities. This includes the object-oriented model of computing as well as agent-based models and the like. The base notion here is that of a specialized structural entity capable of reacting to inputs and, where appropriate, generating appropriate outputs, which it may direct to other structural entities. In contrast to behavior-dominant models, a structural entity may include state information, which is a function of previous inputs and which is maintained throughout the lifetime of the entity.

In practice, it is often necessary to mix these categories in various ways, although, in most cases, one of them represents the dominant paradigm in the sense that it provides the top-level context for the others. For example, object-oriented computing has a structure-dominant model, but the implementation of object operations (methods) is typically procedural (i.e., behavior dominant).

Choosing a dominant model of computation requires both domain expertise and computing technology expertise. The domain experts supply the desired dynamic semantics, usually, in some informal manner, whereas the computing experts select the model or models of computation that most closely map to the desired semantics. The computer experts are also responsible for defining the more formal mapping of the domain semantics to the model of computation.

From this, we can see that there are two principal activities related to the semantics of a modeling language: (1) the design and formulation of the desired semantics and (2) their specification in a way that can be communicated to both humans and computers. Informal but structured and precise specifications of semantics are best suited for human consumption, while, formal, often mathematical, specifications are necessary for computers.

Language designers have a number of choices for specifying the dynamic semantics of their languages. These may be mathematically formal or, as is often the case, specified in the form of computer programs written in some programming language.

[8] The on-line encyclopaedia, Wikipedia, cites the following definition: "… a **model of computation** is the definition of the set of allowable operations used in computation...." (http://en.wikipedia.org/wiki/Model_of_computation).

The latter approach is interesting since such specifications are more easily verified[9] and may also provide the basis for an implementation. In the following section we provide a brief overview of one prominent practical example, from the author's experience, of how the semantics of a relatively large modeling language have been specified.

7.2 The Executable UML Foundation Specification

The Executable UML Foundation [11] is a technology recommendation recently adopted by the OMG, the same industry consortium that manages the UML and MOF language specifications. It is intended to serve two primary purposes:

1. To provide a formal specification of the dynamic semantics of a core subset of UML 2. This subset comprises a complete modeling language in its own right and is the foundation on which the rest of UML 2 is based. The intent is to use this subset as a basis for defining the more complex and higher level elements of UML, such as statecharts and composite structures as shown in Fig. 7.
2. To serve as a standardized facility for describing the semantics of modeling languages in general. Namely, the semantics of a modeling language can, in principle, be defined by specifying them as a program written using the Executable UML Foundation modeling language. (In fact, the Executable UML Foundation is itself the first example of this capability, since it is specified using itself – a common approach in computer language design.)

The Executable UML Foundation specification uses the latter approach to describe the dynamic semantics of the Executable UML Foundation modeling language. This language—a proper subset of the full UML 2 modeling language—is officially called *fUML* (for "foundational" UML). Its semantics are defined by a family of virtual machines capable of executing fUML programs (Fig. 7). For example, the semantics of UML 2 statecharts could be described by writing an appropriate fUML program.

The fUML family of virtual machines is based on a structure-dominant object-oriented model of computation. This is not surprising given that UML is firmly founded on object-oriented principles; it is an example of how the domain can influence the selection of the model of computation. From the modeler's point of view, all behavior in a fUML virtual machine stems from the behavior of application objects responding to messages generated by other application objects sent over links that connect them. These objects exist within the virtual machine environment, which is responsible for their creation and destruction, for performing the transport of messages, for scheduling and dispatching of method executions, and for interactions with entities outside the virtual machine. fUML objects are active objects, which means that they only respond to external events when they perform an explicit receive operation. This allows the modeling of parallel and distributed computations.

[9] Mathematically inclined people may disagree with this conclusion. However, it is the author`s experience that the formal semantics specifications of most computer languages are so complex that their mathematical specifications run on for many pages making them very difficult to verify conclusively.

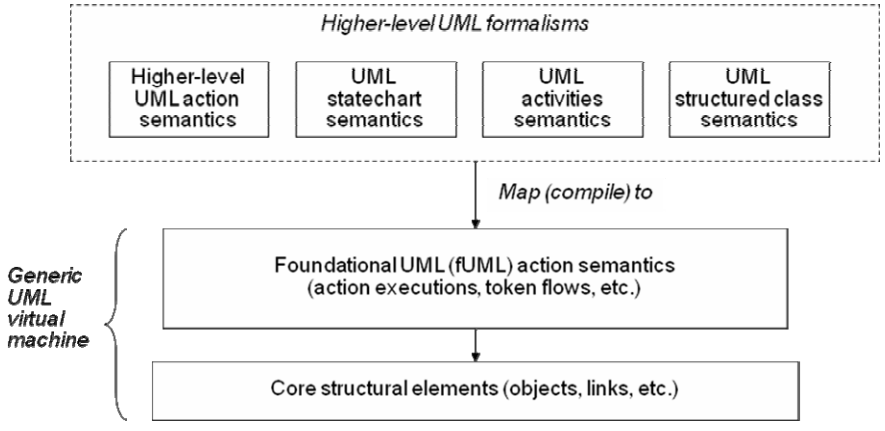


Fig. 7. The Architecture of the Executable UML Foundation

While the dominant model of computation of the fUML virtual machine is structural, their behavior is specified using a flow-based style. fUML objects execute UML actions, whose semantics are described using a combination of data flow (functional) and control flow (procedural) models of computation. A major part of the fUML specification consists of descriptions of the semantics of individual fUML actions (e.g., send message, read attribute, create object). The approach here is based on defining a corresponding "execution" object for each supported action. For example, for the TestIdentity action of UML (which tests whether two different object ids represent the same object), the fUML specification defines a TestIdentityExecution object. Execution objects are created dynamically by the virtual machine and executed when their turn comes. After completing its action, an execution object is no longer useful and is discarded. If execution reaches the same point in the program again, a new execution object will be created.

The specification of how execution objects behave is defined using a proper subset of fUML, called *base UML* or, *bUML*, for short[10]. The reason for using only a subset of fUML to describe itself is that it simplifies the next level of semantics specification. Being simpler than fUML, bUML is easier to define in a formal manner. The ultimate semantics specification in this chain, the specification of bUML, is done

[10] Actually, the current version of the Executable UML Foundation specification uses a strictly controlled subset of the Java programming language (which has relatively well understood semantics) to define the semantics of fUML actions rather than bUML. This subset of Java is chosen in such a way that its statements can be easily and relatively directly mapped to bUML (these mappings are actually included in the spec). The reason for this shortcut was purely practical and was necessitated by the desire to produce a reference implementation that was used to verify the fUML definition. With no fUML or bUML implementation available, some other means had to be found to bootstrap the process. Furthermore, given that there was no textual syntax available for writing bUML programs only graphical representations could be used. Unfortunately, these diagrams tended to be extremely cumbersome and practically unreadable. The full definition of both fUML and bUML is awaiting the definition of a textual version of the UML action language, which is currently under development.

using a mathematical formalism called the *Process Specification Language (PSL)* [3]. This uses an axiomatic approach based on first-order logic to describe the semantics of processes in execution. The PSL specification of bUML serves not only to define the semantics in a mathematically precise fashion, it also opens up the possibility of using computer-based automated reasoning to analyze programs written in bUML, and, ultimately, fUML and UML itself.

Since UML has numerous semantic variation points, some of this variability penetrates down into the semantics of fUML itself. For example, UML leaves the precise semantics of inter-object communications undefined. Thus, it leaves open the question of whether or not messages are delivered in the order in which they are sent (e.g., in priority based communications systems). Consequently, fUML also has semantic variation points, although a much smaller number compared to UML, which is why it is referred to as a "family of virtual machines". This variability allows support for different flavors of executable UML languages.

The feasibility and practicality of using fUML as a general-purpose tool for defining the semantics of modeling languages remains to be tested. However, assuming that most domain-specific languages will likely be simpler than UML, we can be optimistic about its outlook. Ideally, it should be possible to supplement the abstract syntax definition (MOF or profile) with a fUML specification of its dynamic semantics. Given that these are based on standards, it is possible in principle to exchange complete computer-readable language specifications, which can be supported by generic customizable language tools and, in this way, avoid the difficult problem of the missing language support structure.

8 Conclusions and a Look to the Future

In this paper, I have provided a summary of some of the practical and theoretical issues related to design of modeling languages, based on personal experience with MOF, UML, and related languages. The heavy emphasis on these standards should not be interpreted as an endorsement of these languages as paragons of modeling language design. But, whatever their failings (and these are undoubtedly numerous), they are very widely used in both industry and research. This makes them significant by default. As a direct participant in the definition of these standards, I had hoped to provide a candid and unmediated description of the lessons learned and experience gained in the design of these important languages. It still remains, however, for someone to codify and generalize these lessons and incorporate them into a useful and systematic theory of modeling language design that would serve as the foundation of a proper and reliable engineering discipline.

There are indeed many research challenges still facing us before we reach this goal. For example, what should a good metamodeling language look like? EMF is extremely simple compared to the MOF. Which of these approaches is the better one, or, is something in the middle best? Perhaps the answer to that question depends on the type of language being defined; if so, what is the nature of that dependency? And, what about the profile approach, with its promises of minimizing language support structure issues? Under what circumstances is it effective? What should its refinement

constructs look like? Also unresolved is the best way (or ways) in which language semantics are to be defined.

The question that is very much neglected at the moment is the problem of defining concrete syntaxes for modeling languages, particularly graphical ones, although a recently published work shows great promise [8]. Since one of the primary goals of a concrete syntax is to facilitate communications, it surely needs some input from cognitive psychology experts with their insights into human intuition.

There is, of course, the whole topic of model transformations and automated code generation from modeling languages, which has not been touched on in this overview. There are numerous conferences and workshops and many publications on this topic, so at least this issue is not being neglected and some major progress is being made. Surprisingly, however, (at least to me) very little of this work is inspired by the existing and much proven body of knowledge in compiler construction.

Another related issue is that of tools: the current generation of tools to support modeling languages is still immature. Almost all of the important tools are far too complex, fraught with major usability problems, which divert the users' attention from the problem at hand. How should we design our tools to make them flexible and easily adaptable to specific problems and individual needs. How do we construct tools that are capable of adapting to different languages and semantics?

The research topics above represent merely a sampling; the full list is much longer and growing. At the same time, the need for a solid theoretical underpinning is becoming ever more pressing. Currently, it seems to me that industrial practice is far ahead of the theory in this domain, but these solutions tend to be product-specific or project-specific point solutions. These need to be studied and consolidated into general solutions.

In conclusion, there is at present much opportunity for very interesting and very fruitful research in the area of modeling language design. Unfortunately, the current efforts tend to work mostly in isolation of each other, with much relearning and reinvention. I am convinced that this can only be rectified if some form of consolidation of research efforts is undertaken.

References

1. Budinsky, F., Brodsky, S., Merks, E.: Eclipse Modeling Framework. Pearson Education, London (2003)
2. Greenfield, J., et al.: Software Factories. John Wiley & Sons, Chichester (2004)
3. International Standards Organization (ISO), Industrial automation systems and integration–Process specification language, ISO standard 18629 (2005), `http://www.iso.org/iso/home.htm`
4. Kelly, S., Tolvanen, J.-P.: Domain-specific Modeling. IEEE Computer Society Publications, John Wiley & Sons, Inc. (2008)
5. Kleppe, A.: Software Language Engineering—Creating Domain-Specific Languages Using Metamodels. Addison-Wesley, Reading (2008)
6. Mernik, M., Heering, J., Sloane, M.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4), 316–344 (2005)
7. Meyer, B.: UML: The Positive Spin. American Programmer (1997)

8. Moody, D.: The 'Physics' of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. IEEE Transactions of Software Engineering 35(6), 756–779
9. Nunes, N.J., et al.: UML Satellite Activities 2004. LNCS, vol. 3297, pp. 94–233. Springer, Heidelberg (2005)
10. Object Management Group (OMG), Meta Object Facility (MOF) Core, v.2.0, Document formal/06-01-01 (2006)
11. Object Management Group (OMG), Semantics of a Foundational Subset for executable UML Models, Document ptc/2009-10-05 (2009), http://www.omg.org/spec/FUML/1.0
12. Object Management Group (OMG), Unified Modeling Language (UML) Superstructure Specification, v.2.1.1, document formal/07-02-05 (2007)
13. Selic, B., Ward, P.T., Gullekson, G.: Real-Time Object-Oriented Modeling. John Wiley & Sons, Chichester (1995)
14. Selic, B.: Personal Reflections on Automation, Programming Culture, and Model-Based Software Engineering. Automated Software Engineering 15(3-4), 379–391 (2008)
15. Vitruvius, M.P.: The Ten Books on Architecture. Dover Publications, Mineola (1960)
16. Weigert, T., Weil, F.: Practical Experience in Using Model-Driven Engineering to Develop Trustworthy Computing Systems. In: Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing, June 5-7, vol. 1, pp. 208–217 (2006)

Code Transformations for Embedded Reconfigurable Computing Architectures

Pedro C. Diniz[1] and João M.P. Cardoso[2]

[1] Departamento de Engenharia Informática,
Instituto Superior Técnico/INESC-ID,
Av. Prof. Dr. Cavaco Silva,
2780-990 Porto Salvo, Portugal
[2] Departamento de Engenharia Informática,
Faculdade de Engenharia (FEUP), Universidade do Porto,
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal

Abstract. Embedded Systems permeate all aspects of our daily life, from the ubiquitous mobile devices (*e.g.*, PDAs and smart-phones) to play-stations, set-top boxes, household appliances, and in every electronic system, be it large or small (e.g., in cars, wrist-watches). Most embedded systems are characterized by stringent design constraints such as reduced memory and computing capacity, severe power and energy restrictions, weight and space limitations, most importantly, very short life spans and thus strict design cycles. Reconfiguration has emerged as a key technology for embedded systems as it offers the promise of increased system performance and component number reduction. Reconfigurable components can be customized or specialized (even dynamically) to the task at hand, thereby executing specific tasks more efficiently leading to possible reductions of the weight and power. In this article, we introduce and discuss compilation techniques for reconfigurable embedded systems. We present specific compiler techniques focusing on source-level code transformations highlighting their potential and the applicability of generative programming techniques to this compilation domain.

1 Introduction

Reconfigurable computing architectures are playing a very important role in specific computing domains [1]. In the arena of high-performance computing (HPC), Field-Programmable Gate-Arrays (FPGAs) [2] have exhibited in many cases outstanding performance gains over traditional von-Neumann based computer architectures [3]. In the context of embedded systems, FPGAs are common-place for early prototyping, and in deployment, given their unique the ability support "zero-cost" updates of hardware in early product insertion windows where modifications are required, as well as the low initial development costs when compared to ASIC (Application-Specific Integrated Circuit) solutions. The substantial increase of resource capacity in high-end FPGAs and their extreme flexibility has enabled reconfigurable architectures to embrace both traditional and newer markets such as embedded and high-performance computing.

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 322–344, 2011.

The increasingly complexity of reconfigurable architectures and the constant time-to-market pressures have exacerbated the need for the ability to program such architectures at higher-levels of abstractions other than the de facto standard low-level hardware-oriented programming environments. A substantial issue that hampers the mapping of high-level programming languages to reconfigurable architectures is the diversity of granularity of hardware resources in these architectures. Fine-grained reconfigurable fabrics (such as FPGAs) are able to implement digital architectures at a fairly low-level while coarse-grained reconfigurable fabrics are better suited to implement computations based on ALU and storage elements. While coarse-grained architectures offer a set of target programming abstractions such as instructions and memory, they are less flexible than fine-grained architectures in terms of low-level control and scheduling. They facilitate some of the aspects of high-level compilation while hampering the development of very customized low-level digital controller solutions. Fine-grained architectures represent the opposite end of the programmability spectrum. They offer a wide range of design choices but offer virtually no low-level hardware abstractions. Compilers must thus structure their designs in a mix of instructions and dedicated controllers and logic circuits, substantially increasing the complexity of the mapping.

For several years, there have been large efforts on compiling high-level programming languages to reconfigurable architectures [4, 5] and while research has proven that it is possible to bridge the gap between traditional high-level software programming models and hardware-oriented programming languages (*e.g.,* VHDL or Verilog) the extreme low level abstractions exposed by reconfigurable architectures make this compilation problem extremely challenging.

This chapter provides a tutorial on compilation techniques for embedded systems with a special emphasis on architectures consisting of reconfigurable hardware as accelerator units of general-purpose processors. A typical compilation flow, as depicted in Fig. 1, is split in two different compilation processes, one for the segment of the input program to be mapped to the traditional processor and another process corresponding to the segment to be mapped to the hardware accelerator. The hardware compilation process may exhibit different approaches depending on the target hardware accelerator architecture. Depending on the nature of the hardware accelerator, the flow might have to include hardware synthesis steps such as RTL (Register-Transfer-Level) logic synthesis and placement-and-routing (P&R), which in turn requires the generation of hardware-oriented description in languages as interim steps in the compilation flow.

A common approach to reduce the complexity of the compilation process relies on a set of predefined architecture or architectural elements (modules) that are then embedded in the target configurable architecture as soft hardware macro elements. In this approach, the compilation flow needs to include the generation of the macroinstructions to program the hardware elements, possibly requiring a limited form of P&R for the assignment of instructions to various macro elements that will orchestrate the flow of data in the overall hardware design.

This chapter is organized as follows. Section 2 briefly introduces embedded reconfigurable architectures. Then, in Section 3, we describe the main concepts on compilation for reconfigurable architectures. In Section 4, we outline various code transformations along with illustrative examples. Section 5 highlights the potential of

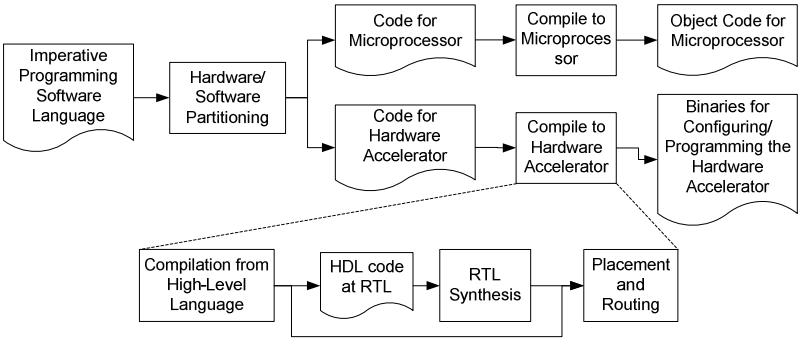


Fig. 1. Typical compilation flow for embedded systems consisting of a microprocessor with hardware accelerators

using generative programming techniques and tools to improve compilation to embedded and reconfigurable architectures. Finally, Section 6 concludes this chapter.

2 Embedded and Reconfigurable Architectures

Reconfigurable computing fabrics – FPGAs being one of the most notable examples – consist mainly of aggregations of a large number of elements, namely: functional units (FUs), memory elements (MEs), interconnection resources (IRs), and I/O buffers (IOBs). Contemporary FPGAs combine fine-grained with coarse-grained elements such as multipliers, DSP blocks, memory blocks and even microprocessors, as is the inclusion of IBM PowerPC cores in some Xilinx FPGA devices. Fig. 2 depicts a possible block diagram of a computing system implemented in a reconfigurable computing fabric.

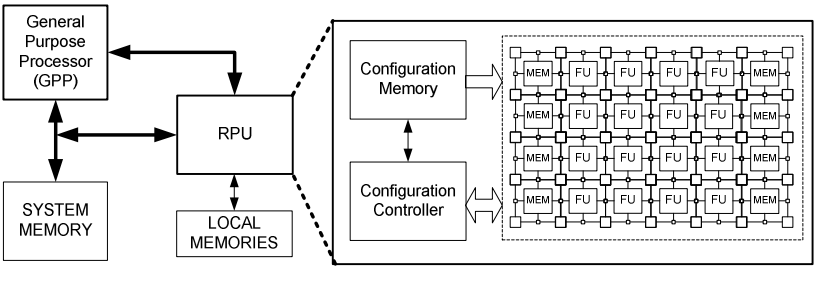


Fig. 2. An example of a possible computing system implemented in a reconfigurable fabric, which includes a general purpose processor (GPP) and a reconfigurable processing unit (RPU)

The reconfigurable fabrics distinguish themselves according to the granularity of the FUs and IRs. Typically, fine-grained architectures include configurable blocks with small bit-widths (*e.g.*, 4 bits), whereas coarse-grained architectures include configurable blocks with large bit-widths (*e.g.*, 16, 24, 32 bits), while other

architectures still expose a mix of fine- and coarse-grained hardware structures. In the case of fine-grained architectures, core cell elements may consist of programmable hardware components that implement Boolean functions with a number of inputs and a single output, as illustrated in Fig. 3(a) for a 2-input fine-grained cell (typical FPGAs use more complex cells with 4 or 6 inputs and 1 or 2 outputs).

Coarse-grained architectures also include a number of cells, (such the one presented in Fig. 3(b)), programmable interconnect resources, and distributed memories. They are more appropriate to implement data-path operations. Note, however, that fine-grained reconfigurable fabrics allow the implementation of microprocessors, configurable processors (*i.e.*, processors with extensible ISA – Instruction-Set Architecture – and parameterizable capabilities [6]) and coarse-grained reconfigurable arrays, as well.

Reconfigurable hardware can be coupled (tightly or loosely) to a processor as a programmable accelerator. In this scenario, the reconfigurable hardware acts as a reconfigurable functional unit implementing extensions to the ISA of the host processor or simply as a co-processor. In this latter case, the reconfigurable hardware can even undertake more sophisticated computations.

The granularity of the reconfigurable fabric constrains the type of computing engines that can efficiently be implemented with its resources. Fine-grained fabrics implement computing engines using gate-level circuitry descriptions (*e.g.*, AND, OR gates), while coarse-grained fabrics implement computing engines at the word or ALU level. Coarse-grained fabrics limit in practice the set of computing models (such as coarse-grain data-flow), whereas in fine-grained fabrics one can implement virtually any type of computing model. This flexibility comes at a cost: programming of fine-grained fabrics is more difficult, cumbersome, and time consuming, imposing a significant overhead in interconnect-resources, to ensure routing between its configurable blocks.

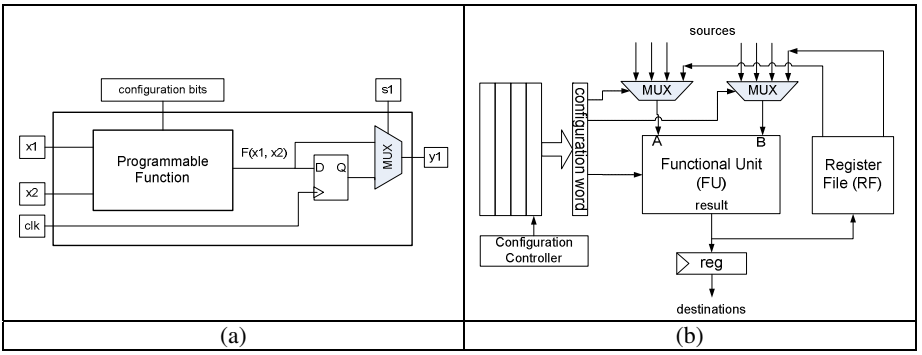


Fig. 3. Example of possible cells: (a) fine-grained; (b) coarse-grained reconfigurable fabrics

One possible approach when compiling high-level programming languages to reconfigurable architectures such as FPGAs relies on the use of hardware mapping templates as the one illustrated in Fig. 4. Here the mapping is decoupled between a control unit and a data-path unit. The control unit is responsible for managing the

execution of the program on the data-path unit, including memory accesses. Both units might be specific (or custom) to a given computation or have programmable features allowing those units to efficiently execute a selected set of programs. An important component of this approach includes the specific use of local data memories and corresponding data partition to the specific needs of the computations.

With respect to the output generated by compilers, it is often based on HDL-RTL descriptions [7] representing computing engines based on the organization depicted in Fig. 4 when targeting fine-grained reconfigurable fabrics, and/or on assembly code to program coarse-grained reconfigurable architectures.

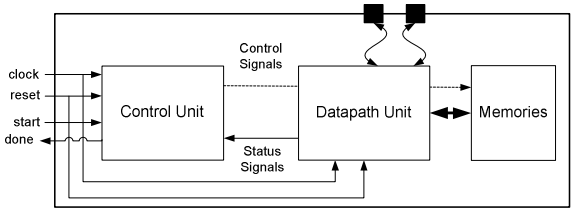


Fig. 4. Typical organization of a computing engine: memory, data-path and control unit

3 Compilation to Reconfigurable Computing Architectures

We now describe key concepts in compilation techniques and execution schemes for reconfigurable embedded architectures. It is assumed the reader to have basic knowledge about compiler analyses and mapping techniques [8-10].

3.1 Resource Sharing

For a given input program, a compiler may bind each operation to a distinct functional unit. To save hardware resources, a compiler may use functional units to implement two or more operations. In this case, a functional unit is shared using a time multiplexed scheme [7, 11]. Correct operands are connected to the inputs of the functional units (using control and multiplexers) and the results of the functional units are connected (possibly with registers) to the operations needing them.

Resource sharing has been used extensively by high-level synthesis tools [7] and several scheduling, allocation and binding steps have been proposed. The typical goal of scheduling is to minimize the number of hardware resources used, possibly via resource sharing, while minimizing the overall execution time. Besides functional units, there are a number of other hardware resources that may benefit from resource sharing, as is the case with memories and input/output ports. In these latter cases, the compiler may not have the opportunity to decide about considering sharing or not and may have to contemplate it. Fig. 5 illustrates a typical interface scheme to a single-port memory. When an operation on a shared hardware resource spans multiple execution stages, a compiler may take advantage of pipelining resource sharing whereby operators simultaneously use different stages of the execution on the resource.

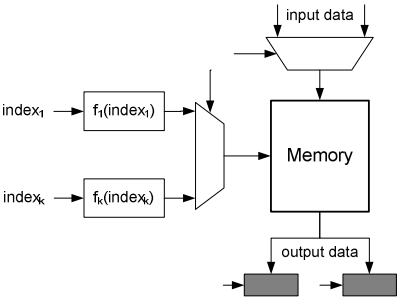


Fig. 5. Interface from different data-path locations to a single-port memory

3.2 Scheduling

Static scheduling defines the control step where a specific operation will be executed. A common scheduling algorithm is the list-scheduling [7, 11] whereby the operations whose input values are already available are scheduled for execution on the set of existing functional units. Two of the more common strategies list-scheduling implementations include the ASAP (As Soon As Possible) and ALAP (As Late As Possible) scheduling information, corresponding respectively to the most eager and the least eager scheduling of operations whose operands have been already computed.

Fig. 6(a) depicts a simple computation expressed as a loop construct in a high-level programming language using array variables. Here we depict the original computation as well as a version where the loop was unrolled by a factor of 2 thereby exposing more basic operations to a scheduling algorithm. In Fig. 6(b) we present the data-flow graph of the unrolled variant of the code and in Fig. 6(c) and (d) we depict the ASAP and ALAP schedule respectively for the original as well as for the unrolled versions of the code.

In these schedules we assume each array variable is mapped to a distinct memory unit and that both memory read and write operations exhibit a latency of 3 clock cycles. We further assume that there is a single multiplier and a single adder unit.

3.3 Loop Pipelining

Loop pipelining is a technique that aims at reducing the execution time of loops by explicitly overlapping computations of consecutive loop iterations. Only iterations (or parts of them), which do not depend on each other can be overlapped. Two distinct cases are: loop pipelining of innermost loops and pipelining across nested loops. Due to its performance benefits, loop pipelining has been the focus of many research efforts and is supported in every major compiler. When compiling to microprocessor-based systems, loop pipelining is known as software pipelining [12]. Many contemporary compilers use as their basis the well-known iterative modulo scheduling technique [12-14]. We illustrate in Fig. 7 a simple example and two possible software pipelining approaches. A first approach is depicted in Fig. 7(a) and uses a kernel set of instructions and explicit prologue and epilogue sections. A second approach, depicted in Fig. 7(b), makes use of predicated instructions with an epilogue and a prologue sections as part of the kernel. The second approach has the advantage

of a lower number of load/store instructions in the code, an important aspect when mapping to architectures where the more the number of load/store instructions in the code higher is the overhead cost.

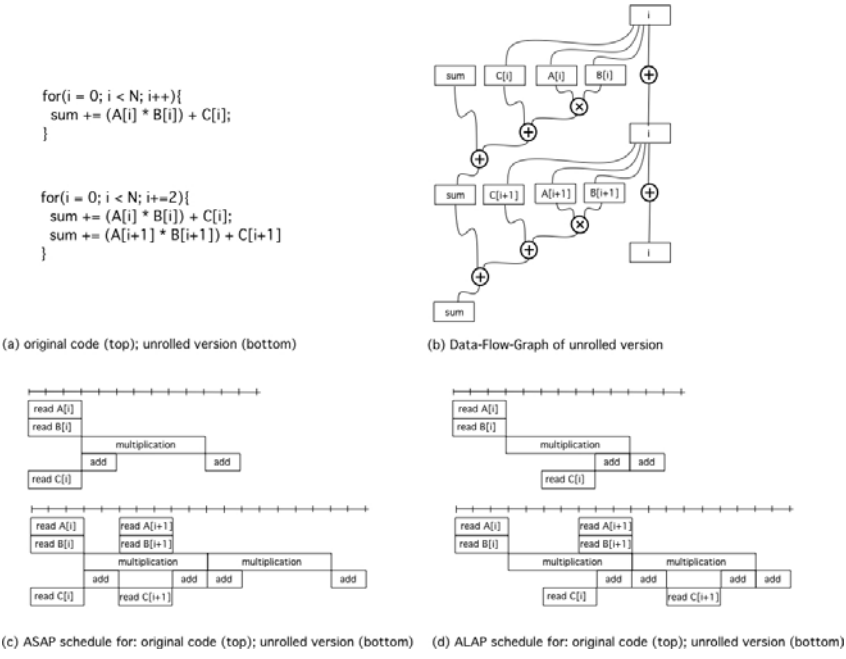


Fig. 6. Scheduling for ASAP and ALAP strategies in the presence of memory accesses for distinct memories and for unrolled loop code versions. In these examples we assume a single multiplier unit and a single adder unit.

for(int i=0; i<10; i++) {
  C[i] = A[i] * B[i];
}
	a_tmp = A[0];
	b_tmp = B[0];
	for(int i=0; i<9; i++) {
	  C[i] = a_tmp * b_tmp;
	  a_tmp = A[i+1];
	  b_tmp = B[i+1];
	}
	C[9] = a_tmp * b_tmp;
		for(int i=0; i<11; i++) {
		  C[i-1] = a_tmp * b_tmp: if i!=0
		  a_tmp = A[i]: if i!=10;
		  b_tmp = B[i]: if i!=10;
		}

(a)	(b)	(c)

Fig. 7. Simple example illustrating software pipelining: (a) original source code; (b) modified source code illustrating the epilogue and the prologue code sections added; (c) use of predication enables software pipelining without explicit epilogue and prologue sections

In the context of reconfigurable or custom architectures, most notably FPGAs, many researchers have exploited loop pipelining in various forms [5]. Some loop transformations (*e.g.*, unrolling, interchange, tiling) can enhance the applicability of loop pipelining. Loop pipelining can be also used to overlap subsequent iterations of an outer loop with an inner loop.

3.4 Task-Level Pipelining

Applications such as video and image/signal processing domains, are naturally structured as sequences of data-dependent tasks using a producer/consumer communication paradigm and are thus amenable to pipelined execution [15, 16]. In this context, one may overlap some of the execution steps of sequences of loops or functions by starting computing as soon as the required data items are produced in a previous function or iteration of a previous loop.

The tasks can communicate data using a consumer/producer model implemented using a custom, application data-driven, fine-grained synchronization buffering. Such an execution scheme allows for out-of-order, data-dependent producer-consumer pairs. Data can be communicated to subsequent stages using a FIFO mechanism [15]. Each FIFO stage may store an array element or a set of array elements. Array elements in each FIFO stage can be consumed by a different order than the one they have been produced. Examples with the same order of producer/consumer only need FIFO stages with one array element. In the other case, each stage must store a sufficient number of array elements in order that all of them are consumed (by any order) before the next FIFO stage is considered. Instead of using coarse-grained (the grain is related to the size of the FIFO stages) synchronization, the approach presented in [16] uses a hash-based inter-stage buffer and signal flags to identify the availability of data elements. Due to the complexity to statically determine the communication needed by this type of pipelining, its application has been limited.

With the widespread use of *multicore* architectures, task-level pipelining may be a key execution technique that exploits the coarse-grained parallelism intrinsic to these architectures. A compiler aware of task-level pipelining must be able to analyze the applicability of this technique, and if applicable it must split the code in tasks and insert the data communication primitives. Reconfigurable hardware resources will bring more opportunities to this task-level pipelining, as special and customized inter-stage buffers can be configured according to the application under execution.

3.5 Temporal Partitioning

Temporal partitioning [17, 18] split a computation in disjoint sections to be executed by time-sharing a hardware resource. Given its similarity to scheduling, various scheduling-based algorithms have been proposed for temporal partitioning. When performing scheduling, in a given control step each operation is assigned to a functional unit trying to execute as many operations concurrently as possible, while still preserving data dependences of the input computation. Between control steps and temporal partitions, data might be communicated typically relying on hardware support in the form of registers, register files, or memories. Fig. 8 illustrates an example of temporal partitioning of a data-flow graph representing a section of an application code. The original data-flow graph shown in Fig. 8(a) is split in three temporal partitions illustrated in Fig. 8(b) communicating data via registers.

There have been many proposed algorithms for temporal partitioning. They consist of extensions to list-scheduling and, *e.g.*, other alternative approaches using optimization algorithms such as simulated annealing [19]. One of the major steps in temporal partitioning algorithms includes the profitability evaluation of the mapping

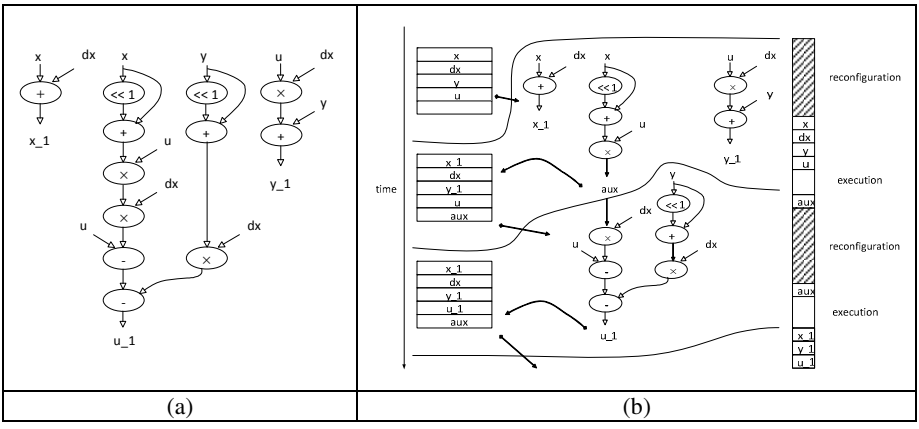


Fig. 8. Example of Temporal Partitioning: (a) data-flow graph representing a section of code; (b) data-flow graph split in three sections executed with time-sharing of the hardware resources (notice the data communicated between sections)

of a given operation to a temporal partition. This step requires an estimation of the resources needed when a specific operation is added to a temporal partition already containing other operations. To simplify this estimation, most proposed schemes use high-level models with algorithmic relaxing (or dampening) factors. Given that temporal partitioning elicits hardware resource sharing several approaches have tried to integrate both problems [20, 21].

3.6 Hardware/Software Partitioning

As most reconfigurable computing platforms include a traditional processor coupled to reconfigurable resources, partitioning the computations between the processor and the reconfigurable resources is a key aspect of the compilation flow. As a result, the original computations are partitioned into a software component and a hardware (reconfigurable) component. The software component is then compiled onto a target processor using a traditional, native compiler for that specific processor. The hardware components are mapped to the reconfigurable resources by, commonly, translating the correspondent computations to representations accepted by hardware compilers or synthesis tools. As part of this partitioning, additional instructions to synchronize the communication of the data between the processor and the reconfigurable resources are required. Fig. 9 illustrates an example of applying hardware/software partitioning over an example and considering a reconfigurable processing unit connected to the microprocessor. This partitioning process is even more complex if the target architecture consists of multiple processors or the communication schemes between devices or cores can be defined in the compilation step, as is the case when targeting complex embedded systems.

Hardware/software partitioning has long been the focus of the hardware/software co-design research community [22]. A common approach relies on the use of migration methods that evaluate the impact of allocating selected segments of code

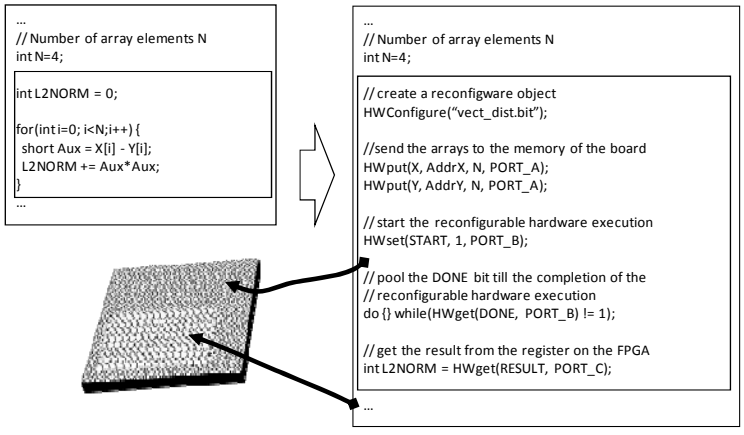


Fig. 9. Migration of an inner loop to reconfigurable hardware (notice the initialization, control and wait loop in the new software version on the right)

from the microprocessor to custom hardware. For instance, the automatic generation of instruction-set extensions (ISEs) supported by dedicated hardware can be thought as a special problem of hardware/software partitioning [23, 24].

4 Code Transformations for Reconfigurable Architectures

We now describe code transformations that are particularly suited for embedded and reconfigurable computing systems. We deliberately exclude from this discussion generic code transformations that compilers routinely perform for all target architectures. For instance, constant folding and constant propagation, dead code elimination, strength reduction or common sub-expression elimination, are often implemented in the compilation process and seldom done explicitly at source-code level. Although most code transformations discussed here are not tied to a particular language, we focus on an imperative programming language such as C as this is a popular embedded systems programming language. In the next subsections we describe a representative set of high-level source-code transformations outlined in Table 1 where we briefly describe them and their perceived benefit.

4.1 Floating-Point to Fixed-Point Precision

It is common that an initial version of an algorithm requiring real numbers is developed using double or single precision data types. After this initial development effort, users typically explore alternative implementations with reduced precision requirements, such as fixed-point representations. Operations over real numbers do not necessarily require dedicated floating-point units thus allowing for lower-cost design solutions. In this process, the algorithms must be adapted to use fixed-point representations commonly performed at source-code level with user help and/or of assistant tools (see, *e.g.*, [25]).

Table 1. Code transformations (transformations indicated by * are described in greater detail in the following sub-sections)

Transformation	Description
Floating- to fixed-point data types and floating-point precision conversions (e.g., double vs. single precision)*	Transforms floating-point to fixed-point numeric representations. Requires analysis – static and/or dynamic – for deciding the integer and fractional representation components. A related transformation includes conversion to non-standard floating-point representations (*i.e.*, non IEEE 754). Similar transformations recast computations using higher-precision floating-point representations such as single to double precision.
Loop transformations (e.g., loop unrolling, distribution, fusion, fission, reorder)	Loop-based transformations from loop unrolling (partial or full), loop distribution, loop fission, or combinations thereof such as unroll-and-jam. Overall increase opportunities for ILP and other forms of concurrency such as task parallelism.
Software Pipelining	Transforms a loop in order to have operations of subsequent iterations overlapped with the current iteration. This transformation might explicitly use prologue and epilogue or the use of predicates.
Data distribution*, Custom data layout* and Data replication*	Divides/Replicates datasets in chunks in order to distribute them, possibly in a custom fashion, by multiple memories in the system thereby enabling concurrent access to multiple data items.
Scalar promotion (elimination of memory accesses)	Converts array elements into scalar variables eliminating the need for array indexing and in many cases memory accesses. When combined with replication and distribution can increase data availability and eliminate anti-dependences.
Data reuse* (elimination of redundant memory accesses)	Transforms a computation so that data (usually in array variables) that are frequently accessed can be cached (typically in registers) thereby eliminating many accesses to array variables. This transformation is commonly implemented using scalar variables, rotating registers, shift registers, and local memories in conjunction with loop unrolling and constant propagation.
Function inlining and exlining	Replaces function calls by the function body (inlining) or code patterns or clones to calls to functions (exlining) having as body such patterns or clones.
Code motion (hoisting)	Moves code around basic blocks, loops, and functions. A special is loop-invariant code motion that movies code outside loops.
Pointers to array variables	Transforms pointers to array variables. This might improve data-dependence analysis and as consequence better results.

There are two common cases of fixed-point representations: (a) uniform fixed-point data type (*i.e.*, all the variables use the same number of bits to represent the fractional and the integer fields); (b) non-uniform (variable) fixed-point representation for each variable (*i.e.*, each variable may have a different number of bits for integer and fractional fields). Fig. 10 illustrates these two cases. The original code in Fig. 10(a) uses floating-point data types. This code can be transformed to a uniform fixed-point representation taking advantage of macros as illustrated in Fig. 10(b), using in this case 32 bits of word-length and 16 bits for the fractional field. A non-uniform (variable) fixed-point representation requires more code modifications to specify for each operation, the properties of each operand and the corresponding result (as is shown in Fig. 10(c) for some code statements).

4.2 Data Reuse and Scalar Replacement

Computations often reuse data values, for example when they repeatedly use coefficients of a given signal transformation or when they repeatedly access, possibly in an overlapped fashion, a region of an array variable. An implementation can exploit this data reuse by selectively choosing which data values to reuse, caching them in

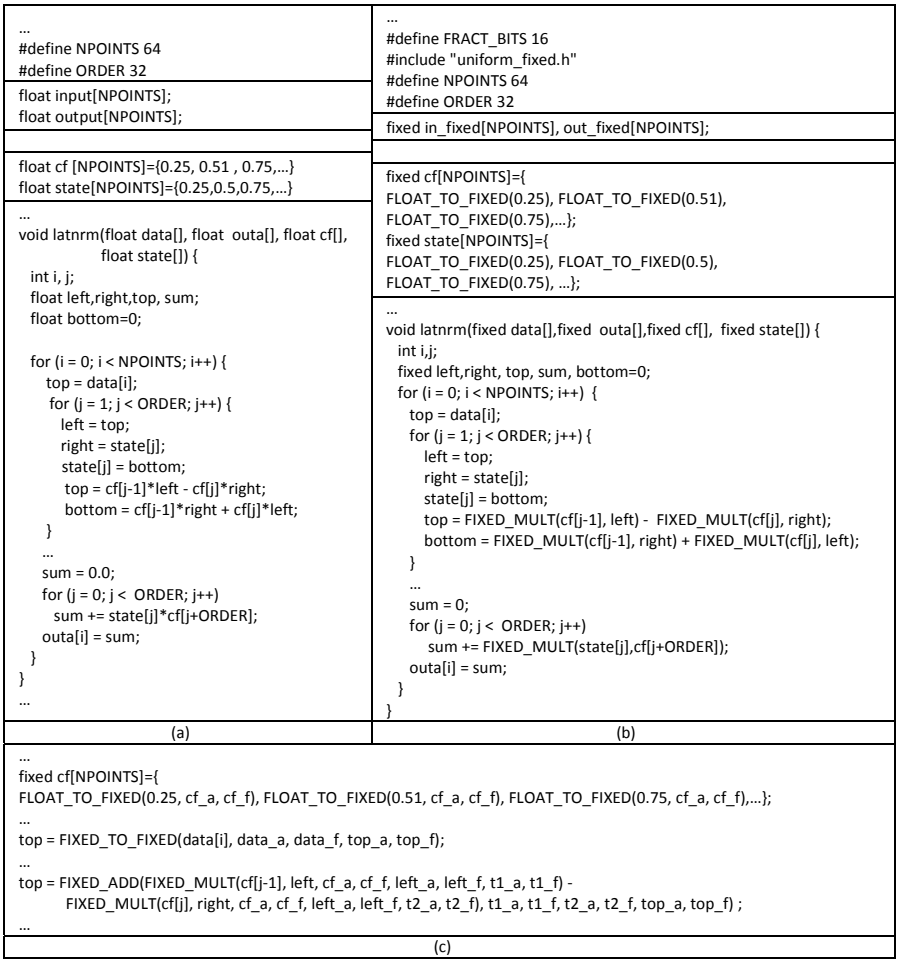


Fig. 10. Examples of floating- to fixed-point data representation conversion: (a) original code; (b) code using macros to represent operations on fixed-point data; (c) code sketch with non-uniform fixed-point data type representations

scalar variables the first time they are accessed. These values, saved into registers or internal memories, are then reused in the remainder of the computation until they are no longer needed. As with traditional architectures, this caching of data "locally" in registers or memories improves the overall hardware implementation performance as it reduces data access latency in many cases even eliminating memory accesses [26].

Fig. 11 illustrates the application of this data reuse concept using a transformation called scalar replacement. Here successive iterations of the loop in Fig. 11(a) access an overlapping set of data values. At each loop iteration, two of the three values used have been accessed in the previous iteration. A way to exploit this reuse is as shown in Fig. 11(b). The code uses three scalar variables `x_0`, `x_1` and `x_2` and shifts the values through them at each iteration. Before the loop executes, the first two values of

```
for (int i = 2; i< N; i++) {
  y[i] = x[i] + x[i-1] + x[i-2];
}
```

(a)

```
int x_2 = x[0]
int x_1 = x[1];
Int x_0;
for (int i = 2; i< N; i++) {
  x_0 = x[i];
  y[i] = x_0 + x_1 + x_2;
  x_2 = x_1;
  x_1 = x_0;
}
```

(b)

Fig. 11. Transformations for data reuse: (a) original code; (b) transformed code

the array `x` are fetched to the scalar variables. At each iteration of the transformed code a single array access is performed, a substantial reduction from the original code.

As can be observed by this example, this data transformation has the potential to substantially reduce the number of external memory accesses at the expense of local storage in the form of registers or internal memory storage. Researchers have developed sophisticated compiler data dependence analyses and code transformations to derive efficient hardware implementations that exploit these reuse opportunities [27].

4.3 Data Distribution and Custom Data Layout

Unlike the previous transformation, data distribution increases the availability of data by distributing the data through distinct memories. Data distribution, commonly used for array variables, partitions the array data into disjoint subsets of data, each of which is mapped to a distinct memory unit. When used in combination with loop unrolling, data distribution, allows the generation of hardware implementations that concurrently accesses data without memory access contention.

Fig. 12 illustrates the application of loop unrolling and data distribution for the `img` array of the example code. As an interim transformation step, the original `img` array is first partitioned into two distinct arrays, `imgOdd` and `imgEven`, bound to two different memories. This distribution then allows the two simultaneously memory loads corresponding to the unrolled statement in the code in Fig. 12(b).

As it is apparent, data distribution does not increase the required storage needs as the original data is partitioned into disjoint data sets. Other than a possible execution

```
...
type img[N][N];
...
for(j=0; j < N; j++) {
  ...
  for(i=0; i < N; i++) {
    ... = img[j][i];
  }
  ...
}
```

(a)

```
...
type imgOdd[N][N/2], imgEven[N][N/2];
...
for(j=0; j < N; j++) {
  ...
  for(i=0; i < N; i+=2) {
    ... = imgOdd[j][i/2];
    ... = imgEven[j][i/2];
  }...
}
```

(b)

Fig. 12. Loop unrolling and array data distribution example: (a) Original C source code; (b) Loop unrolled by a factor of 2 and distribution of the `img` array variable

time overhead in data reorganization via distribution, this transformation increases the availability of data provided the underlying architecture has enough memory modules with adequate capacity to accommodate the data.

As with other array-based computations, researchers have developed sophisticated data dependence analysis techniques that can derive a custom data layout to the data access pattern of array variables of a computation, typically in a loop nest, and maps section of these arrays to distinct memories to minimize memory access time [26].

4.4 Data Replication

Unlike the data distribution and data layout transformations described previously, data replication creates various copies, or replicas, of specific data values in distinct storage structures. Data replication thus increases the availability of the data and thus allows concurrent data accesses. Fig. 13 illustrates the application of loop unrolling, and data replication for the `img` array of the example code. The replication creates two copies of the `img` array, respectively `imgA` and `imgB` that are matched to the array data accesses resulting in the unrolled code for the inner loop.

```
...
type img[N][N];
...
for(j=0; j < N; j++) {
  ...
  for(i=0; i < N; i++) {
    ... = img[j][i];
  } ...
}
```

(a)

```
...
type imgA[N][N], imgB[N][N];
...
for(j=0; j < N; j++) { ...
  for(i=0; i < N; i+=2) {
      ... = imgA[j][i];
      ... = imgB[j][i+1];
  } ...
}
```

(b)

Fig. 13. Loop unrolling and array data replication example: (a) Original C source code; (b) Loop unrolled by 2 and replication of `img`

This transformation has to be exercised with caution as it increases the availability of data at the expense of potentially substantial increase in allocated storage space. As a result, its applicability may be limited to small array variables that are immutable throughout the computation, *i.e.*, to variables that hold coefficient or parameter data that need to be accessed frequently and freely. In addition to these storage issues, there is also the issue of consistency should the data be modified during the computation. In this case, the execution must ensure all replicas are updated with the correct values before the copies can be accessed [27].

4.5 Combining Source-Level Code Transformations

We now illustrate the impact of source code transformations using as key performance metrics the number of external memory accesses. We use as an illustrative example the *smooth* image-processing operator depicted in Fig. 14(a) and hardware architectures suggested by the application of the various transformations as illustrated in Fig. 16. In these hardware architectures we explore the existence of multiple local memories and the possibility of simultaneously memory accesses.

In this example, the computation performs a convolution of a 3-by-3 window with the values stored in the `K` array. This convolution is implemented by the two inner loops on respectively the index variables `r` and `c` (standing for row and column) that accumulate the value in the `sum` variable. After normalization, each sum value is stored in the output image at a pixel location corresponding to the center of the window of the input image used. The computation is repeated for the adjacent windows both "below" (in the sense of increasing value of the index `i` variables) and to the "right" (in the sense of the increasing value of the `j` variable).

```
1.  int sizeX= 350;
2.  int sizeY= 350;
3.  void smooth(short[][] IN, short[][] OUT){
4.    short[][] K = {{1, 2, 1},{2, 4, 2},{1, 2, 1}};
5.    for(int j=0; j < sizeY-2; j++)
6.      for(int i= 0; i < sizeX-2; i++) {
7.        int sum = 0;
8.        for(int r=0; r < 3; r++)
9.          for(int c = 0; c < 3; c++)
10.           sum += IN[j+r][i+c]*K[r][c];
11.       sum= sum / 16;
12.       OUT[j+1][i+1] =  (short) sum;
13.     }
14. }
```

(a)

```
1.    int sizeX= 350;
2.    int sizeY= 350;
3.    void smooth(short[][]IN, short[][] OUT) {
4.    for(int j=0; j < sizeY-2; j++)
5.      for(int i= 0; i < sizeX-2; i++) {
6.        int sum += IN[j][i];
7.        sum += IN[j][i+1]*2;
8.        sum += IN[j][i+2];
9.        sum += IN[j+1][i]*2;
10.       sum += IN[j+1][i+1]*4;
11.       sum += IN[j+1][i+2]*2;
12.       sum += IN[j+2][i];
13.       sum += IN[j+2][i+1]*2;
14.       sum += IN[j+2][i+2];
15.       sum= sum / 16;
16.       OUT[j+1][i+1] =  (short) sum;
17.     }
18.  }
```

(b)

Fig. 14. *Smooth* operator: (a) original code; (b) transformed code using loop unrolling and scalar replacement

A naïve implementation of this computation would have all variables mapped to external storage requiring 2,179,872 load operations and 121,104 store operations resulting in a total of 2,300,976 external memory operations. Clearly, and even if adequately pipelined, these operations are simply too many and consume a substantial amount of energy and execution time. An obvious improvement on this computation would consist in the application of loop unrolling to the two innermost loops given that their iteration bounds are small and in the use of scalar replacement of the references to the `K` array. The resulting source code, depicted in Fig. 14(b), performs a total of 1,089,936 external memory load operations, 121,104 external memory store operations or a total of 1,211,040 a reduction of approximately 50% of the external memory operations. There is also potential for even further reduction of the number of memory accesses as subsequent iterations of the `i` loop do indeed use common input values, as 6 out of the 9 values used in the previous iteration of this loop can be reused if cached in local registers or internal memories.

Exploiting this opportunity, we can apply the data-reuse transformation over the `IN` array across both the `i` and the `j` loop. These reuse variants exhibit increasingly levels of data reuse, and thus increasing reduction on the number of memory operations at the expense of increasing internal storage requirements.

In the simplest data reuse variant, we only exploit data reuse in the innermost `i` loop, which corresponds to the reuse along the rows of the `IN` array. We can exploit this reuse at the source level by capturing in the first iteration of the `i` loop (as dictated by the predicate: `i == 0`) where the code loads and saves in the internal memory array `IN_REG`, all the 9 data items fetched for the `IN` array. In the subsequent iterations of the `i` loop, the code only fetches the "leading" 3 data items corresponding to each of the three rows as the "window" moves to the "right" (direction of increasing indices of `j`). The values in `IN_REG` array are reused in subsequent iterations of the loop by rotating the values of the indices `line0`, `line1` and `line2` at the completion of each iteration, requiring an internal memory with 9 positions[1]. Fig. 15(a) depicts this transformed code where we have omitted some of the source code for brevity and space considerations. For this code implementation, we have a total of 365,400 external memory load operations, 121,104 external store operations resulting in a total of 486,504 external memory operations, and thus a reduction of 4.7× over the original code implementation.

Another code variant, can exploit reuse across iterations of the outermost loop, the `j` loop, saving the values read across two consecutive iterations of the `j` loop. The transformed code, depicted in Fig. 15(b), saves the data elements in the `IN` array accessed in the first iteration of the `j` loop and in the first iteration of the `i` loop in an internal memory. In this implementation we have a total of 122,500 external memory load operations, 121,104 external store operations resulting in a total of 243,604 external memory accesses, and thus a reduction of 9× over the original implementation. As each value the computation uses is fetched from external memory only once, this implementation is optimal from the viewpoint of the number of external memory accesses. This implementation, however, comes at the cost of requiring an internal memory with 3 times the number of pixels in each image row, in this case 3×350 values.

As seen by this example, compilation to reconfigurable architectures and embedded systems with multiple local memories requires a vast number of code transformations to allow efficient implementations. Fig. 16 depicts four possible hardware designs suggested by the combination of two of these transformations. In Fig. 16(a) we have the naïve or base hardware architecture for the smooth operator code. Here the core data-path is responsible for the direct hardware implementation of the arithmetic operations in the inner loop of the algorithm implementation.

In Fig. 16(b) we have a design where the input `IN` array has been replicated and distributed among three RAM blocks, each being accessed for a consecutive line of that array. In Fig. 16(c) we have a design suggested by the application of scalar replacement where consecutive accesses to the elements of a row of the `IN` array are drawn from a tapped-delay line. Each of the delay lines thus mimics the movement of the "window" in the smooth operation. Finally, in Fig. 16(d) we have a design

[1] In reality a clever implementation only requires an internal memory with 6 positions as the values of the leftmost column in the 3×3 window can be immediately replaced by other 3 values after being used in the current iteration computation. This implementation "trick" however complicates the code generation substantially as values need to be rotated as they are being used for the last time in the computation.

```
1.  int sizeX= 350;
2.  int sizeY= 350;
3.  int MEM_SIZE = 3;
4.
5.  short IN_REG[MEM_SIZE][MEM_SIZE];
6.
7.  void smooth(short[][]IN, short[][] OUT) {
8.    int line0=0, line1=1, line2=2;
9.
10.  for(int j=0; j < sizeY-2; j++) {
11.    for(int i= 0; i < sizeX-2; i++) {
12.      if (i == 0){
13.        // loading /saving the first 6 elements
14.        // of 3x3 window in internal memory
15.        IN_REG[line0][0] = IN[j][i];
16.        IN_REG[line0][1] = IN[j][i+1];
17.        IN_REG[line1][0] = IN[j+1][i];
18.        IN_REG[line1][1] = IN[j+1][i+1];
19.        IN_REG[line2][0] = IN[j+2][i];
20.        IN_REG[line2][1] = IN[j+2][i+1];
21.      }
22.      // loading/saving last column of 3x3 window
23.      IN_REG[line0][2] = IN[j][i+2];
24.      IN_REG[line1][2] = IN[j+1][i+2];
25.      IN_REG[line2][2] = IN[j+2][i+2];
26.      int sum = IN_REG[line0][0];
27.      sum += IN_REG[line0][1]*2;
28.      sum += IN_REG[line0][2];
29.      sum += IN_REG[line1][0]*2;
30.      sum += IN_REG[line1][1]*4;
31.      sum += IN_REG[line1][2]*2;
32.      sum += IN_REG[line2][0];
33.      sum += IN_REG[line2][1]*2;
34.      sum += IN_REG[line2][2];
35.      sum= sum / 16;
36.      OUT[j+1][i+1] =  (short) sum;
37.      // rotate pointers line0, line1 and line2
38.      line0 = ((line0+1) % MEM_SIZE);
39.      line1 = ((line1+1) % MEM_SIZE);
40.      line2 = ((line2+1) % MEM_SIZE);
41.    }
42.  }
```

(a)

```
1.  int sizeX= 350;
2.  int sizeY= 350;
3.  int MEM_SIZE = 3;
4.  short IN_REG[MEM_SIZE][sizeX];
5.  void smooth(short[][]IN, short[][] OUT) {
6.  int line0=0, line1=1, line2=2;
7.  for(int j=0; j < sizeY-2; j++) {
8.    for(int i= 0; i < sizeX-2; i++) {
9.      if (j == 0){    // loading / saving the
10.       if (i==0) {   // first 3 rows of the IN array
11.         IN_REG[line0][0] = IN[j][i];
12.         IN_REG[line0][1] = IN[j][i+1];
13.         IN_REG[line0][2] = IN[j][i+2];
14.         IN_REG[line1][0] = IN[j+1][i];
15.         IN_REG[line1][1] = IN[j+1][i+1];
16.         IN_REG[line1][2] = IN[j+1][i+2];
17.         IN_REG[line2][0] = IN[j+2][i];
18.         IN_REG[line2][1] = IN[j+2][i+1];
19.           IN_REG[line2][2] = IN[j+2][i+2];
20.       } else {    // (j == 0) and (i != 0)
21.         IN_REG[line0][i+2] = IN[j][i+2];
22.         IN_REG[line1][i+2] = IN[j+1][i+2];
23.         IN_REG[line2][i+2] = IN[j+2][i+2];
24.     } else { // (j != 0)
25.         IN_REG[line2][i+2] = IN[j+2][i+2];
26.     }
27.     // now the smooth computation
28.     int sum = IN_REG[line0][i];
29.     sum += IN_REG[line0][i+1]*2;
30.     sum += IN_REG[line0][i+2];
31.     sum += IN_REG[line1][i]*2;
32.     sum += IN_REG[line1][i+1]*4;
33.     sum += IN_REG[line1][i+2]*2;
34.     sum += IN_REG[line2][i];
35.     sum += IN_REG[line2][i+1]*2;
36.     sum += IN_REG[line2][i+2];
37.     sum= sum / 16;
38.       OUT[j+1][i+1] =  (short) sum;
39.       // rotate pointers line0, line1 and line2
40.       line0 = ((line0+1) % MEM_SIZE);
41.       line1 = ((line1+1) % MEM_SIZE);
42.       line2 = ((line2+1) % MEM_SIZE);
43.     }
44.  }
```

(b)

Fig. 15. *Smooth* operator after loop unrolling and scalar replacement: (a) using internal `IN_REG` array, and (b) using internal memory `IN_REG` array with reuse along the `j`-loop of the computation

suggested by the application of data replication/distribution and scalar replacement. Only in an architecture with multiple configurable resources such as RAM blocks, interconnects and fine-grain registers could these designs be efficiently implemented.

4.6 Significance of Code Transformations in Reconfigurable Computing

The transformation examples described in the previous section highlight the significance of those transformations in the context of configurable and reconfigurable architectures. While many of those transformations can be successfully applied in the context of traditional processor-based architectures (either single or multi-core),

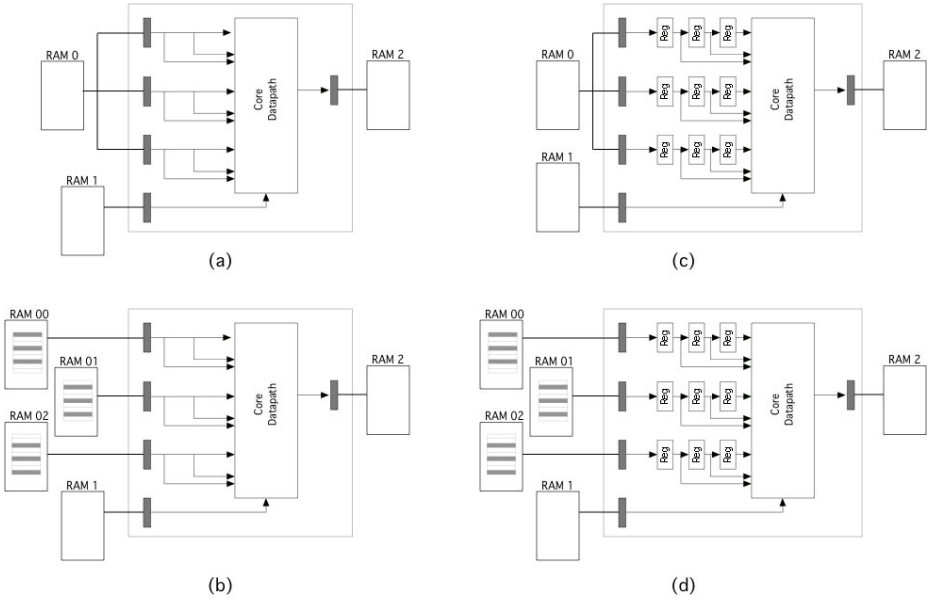


Fig. 16. Possible resultant architectures: (a) with 3 RAM blocks; (b) with 3 RAM blocks and tapped delay-lines; (c) with 4 RAM blocks; (d) with 4 RAM blocks and tapped delay-lines

configurable and reconfigurable architectures magnify, in many instances, their performance impact. We now highlight some key examples of these transformations and describe the specific features of reconfigurable architectures that enable performance (either time, space, power, or energy) gains. In this description we broadly classify source code transformations in three categories, namely:

- **Representation-oriented Transformations:** In this class of transformations, one can include numeric representation transformations that take advantage of the fact that in reconfigurable architectures (most notably FPGA-like architectures) one can select precisely the number of bits for the numeric representation of the values of interest as well as the specific implementation of the operators that manipulate them. As such, each individual operation only manipulates the bits strictly needed resulting in space, time and power savings over pre-defined numeric operators such as the ones using floating-point units. Customization of the numeric values is a key application and/or code transformation in industrial and embedded applications for enhanced accuracy and energy savings (see, *e.g.*, [33]).
- **Computation-oriented Transformations:** These transformations expose additional operations per "synchronization" or "control" point in the computation's execution (*e.g.*, loop iteration). These operations can be executed concurrently in a spatial fashion rather than being pipelined in a single execution unit. In fine-grained reconfigurable architectures one can lay down a custom pipeline structure for each computation to match the computation at hand. Typically, this will involve the use of intermediate storage structures to save temporary (intermediate) results that would otherwise have to be written and read back from a local or global storage.

The profitability and performance increase now depends critically not only on the true control and data dependences of the computation, but also on the ability to feed and schedule the various functional units. When combined with data-oriented transformations, transformations such as loop unrolling and loop fission may create independent tasks of variable granularity that can be executed concurrently.

- **Data-oriented Transformations:** These transformations remap data locations in the traditional address-space organization to physical storage structures. In the case of scalar-replacement and the corresponding mapping of the array values to elements of a tapped-delay line, in a single clock cycle a large number of data transfer can occur. Furthermore, and because the indexing of the elements to be used in the computation is fixed (as they are drawn from the very same tap in the delay line), huge savings in array indexing are accomplished. Some of these transformations, such as data partitioning, have already been explored in the context of distributed-memory multiprocessors. In the context of reconfigurable architectures, however, the richness of the programmability of the underlying interconnection resources allows transformations that would be otherwise less appealing. An example is the use of data replication where in a single clock cycle multiple storage structures can be written making the cost of replication very low.

In all of the transformations highlighted above, the potentially large savings are achieved by the ability of the underlying architecture to be customizable to the specific structure of the computation and/or storage at hand.

4.7 Key Compiler/Design Tool Challenges for Reconfigurable Architectures

Arguably the biggest challenge in leveraging the diversity and wealth of code transformations is the adequate selection of which and in what sequence should a compiler or synthesis tool apply them. For fine-grained reconfigurable architectures, such as FPGAs, this problem is compounded by the fact that additional back-end steps such as placement-and-routing (P&R) need to be carried out with the possibility that the target device does not meet the required resources budget. Thus, selecting the transformations and understanding the interplay between them and the specific nature of the target architecture is of paramount importance for dealing with the potentially huge design spaces these code transformations create.

Exploring these design spaces and generating an implementation for each possible design-space point is clearly infeasible. Instead we believe several key techniques and the corresponding challenges to make their application a reality will be required to ameliorate or mitigate this search, namely:

- **Modeling and Estimation:** If compilers are going to be successful in leveraging the wealth of code transformations and their interactions, adequate performance models for the effects of each transformation in each of the relevant metrics need to be devised [34]. In various contexts, it seems very unlikely that precise modeling is possible. As such, estimation techniques offer a valuable approach to avoid long compilation/synthesis times at the expense of precision loss [35]. Recent experiences in limited contexts have shown that it is possible to use imprecise information and still make correct design choices.

– **Virtualization and Module Generation:** One possible approach to mitigate the complexity and diversity of the target configurable architectures includes the use of virtualization in combination with module generation techniques for code generation. Virtualization would offer an intermediate representation where abstract computation and storage structure could be both described and modeled. The use of module generators integrated with the data-flow graph representing the overall program enables the specialization of each module instance without incurring in long code generation times. The use of pre-placed and pre-routed soft-macros is a key technique in this setting. The key issue, in particular for coarse-grained reconfigurable architectures, lies in the ability not to excessively fragment the overall hardware implementation.

While in general any of these challenges seems insurmountable, exploring selected applications and architectures thereby limiting the diversity of the choices may prove to be a winning strategy. Work in the domain of digital and signal processing has shown that it is possible to attain full automation and thus correct designs with extremely short design times, at the expense of a very modest performance loss when compared to hand-coded designs [36]. In this context, domain-specific languages [37] or the use of aspect-oriented programming [28] where the user can convey key features of the input domain or requirements of the desired solution provide a stepping stone to reduce the compiler's mapping complexity and thus make configurable technology approachable to the average programmer.

5 The Importance of Generative Programming

Source-level code transformations are a key program transformation technique for improving specific execution metrics such as time, energy or space (memory). These transformations, however, often have conflicting goals making their choice and ordering an extremely hard optimization problem. For example, data replication increases data availability (*i.e.*, available bandwidth) at the expense of memory storage. In addition, transformations are very cumbersome and error-prone for programmers to apply them manually, thus suggesting the use of automated code transformation systems and/or compilation tools. To exacerbate these difficulties, many if not all of these transformations require program knowledge that is often neither present in the original program specification nor can it be easily extracted from it. As such, compilers that use them are extremely limited by the semantics of the input programming language and thus require programmer input to guide them in the applicability of these transformations.

Generative programming tools [29] offer a path for the automation of code transformations. They may allow programmers to augment computation with key desired metric goals that will lead to the development of internal transformation application strategies. These approaches thus rely on linguistics mechanisms that are beyond the semantics of the input program languages, typically in the form of rule-based systems (*e.g.*, pattern-apply-condition). We thus believe to be very important to exploit the synergies between the more traditional domains of compiler optimization and code transformations holistic concepts from generative programming. Tools such

as TXL [30], Tom [31], and Stratego/XL [32] may undoubtedly play an important role by enhancing the compilation flow with code transformation rules and strategies.

When integrated in a compilation flow, the use of mature transformation tools such as Tom [31] has several advantages as it may expose to developers a powerful way to derive transformations and to apply different strategies that can be very important to achieve the desired performance.

6 Conclusions

This chapter described key concepts on compilation for reconfigurable architectures, from a base flow to a flow that incorporates source-level program transformations with the aim at improving the custom generated architecture. As reconfigurable architectures support both spatial and temporal computing, they allow aggressive forms of parallelism and concurrent execution. Effective compilation thus requires the synergy of techniques from various domains, from high-level dependence analysis to low-level hardware scheduling. As highlighted we have outlined a possible research direction with the use of generative programming techniques to assist code transformations, which we hope this article has motivated and which we believe is a promising avenue to address and overcome the many programming challenges reconfigurable embedded architectures impose.

Acknowledgments

This work was partially supported by Portugal's "Fundação para a Ciência e Tecnologia (FCT)", under grants PTDC/EEA-ELC/70272/2006 and PTDC/EIA/70271/2006.

References

1. Hauck, S., DeHon, A.: Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation. Morgan Kaufmann/Elsevier (2008)
2. Kuon, I., Tessier, R., Rose, J.: FPGA Architecture: Survey and Challenges, in Foundations and Trends in Electronic Design Automation, pp. 135–253 (2008)
3. El-Ghazawi, T., et al.: The Promise of High-Performance Reconfigurable Computing. Computer 41(2), 69–76 (2008)
4. Cardoso, J.M.P., Diniz, P.C.: Compilation Techniques for Reconfigurable Architectures. Springer, Heidelberg (2008)
5. Cardoso, J.M.P., Diniz, P.C., Weinhardt, M.: Compiling for Reconfigurable Computing: A Survey. ACM Computing Surveys (CSUR) 42(4) (2010)
6. Gonzalez, R.: Xtensa: a configurable and extensible processor. IEEE Micro. 20(2), 60–70 (2000)
7. Gajski, D., et al.: High-level Synthesis: Introduction to Chip and System Design, p. 359. Kluwer Academic Publishers, Dordrecht (1992)
8. Muchnick, S.: Advanced Compiler Design and Implementation, p. 856. Morgan Kaufmann Publishers Inc., San Francisco (1997)

9. Wolfe, M.: High Performance Compilers for Parallel Computing. In: Carter, S., Leda, O. (eds.), p. 570. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (1995)
10. Aho, A., et al.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison Wesley, Reading (2006)
11. Micheli, G.: Synthesis and Optimization of Digital Circuits. McGraw Hill, New York (1994)
12. Vicki, A., et al.: Software pipelining. ACM Computing Surveys (CSUR) 27(3), 367–432 (1995)
13. Haldar, M., et al.: A System for Synthesizing Optimized FPGA Hardware from MATLAB. In: IEEE/ACM Intl. Conf. on Computer-Aided Design (ICCAD 2001). IEEE Press, San Jose (2001)
14. Gokhale, M., Stone, J., Gomersall, E.: Co-Synthesis to a Hybrid RISC/FPGA Architecture. Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology 24(2), 165–180 (2000)
15. Ziegler, H., et al.: Coarse-Grain Pipelining on Multiple FPGA Architectures. In: 10th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM 2002). IEEE Computer Society Press, Los Alamitos (2002)
16. Rodrigues, R., Cardoso, J.M.P., Diniz, P.C.: A Data-Driven Approach for Pipelining Sequences of Data-Dependent Loops. In: Proc. of the 15th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM 2007). IEEE Computer Society Press, Los Alamitos (2007)
17. Purna, K., Bhatia, D.: Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers. IEEE Trans. on Computers 48(6), 579–590 (1999)
18. Ouaiss, I., et al.: An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures. In: Proc. 5th Reconfigurable Architectures Workshop (RAW 1998). Springer, Orlando (1998)
19. Cardoso, J.M.P., Neto, H.C.: An Enhanced Static-List Scheduling Algorithm for Temporal Partitioning onto RPUs. In: IFIP TC10/WG10.5 Tenth Intl. Conf. on Very Large Scale Integration (VLSI 1999). Kluwer Academic Publishers, Lisbon (2000)
20. Cardoso, J.M.P.: On Combining Temporal Partitioning and Sharing of Functional Units in Compilation for Reconfigurable Architectures. IEEE Trans. on Computers 52(10), 1362–1375 (2003)
21. Pandey, A., Vemuri, R.: Combined Temporal Partitioning and Scheduling for Reconfigurable Architectures. In: SPIE Photonics East Conference. SPIE - The International Society for Optical Engineering, Boston (1999)
22. De Micheli, G., Gupta, R.: Hardware/Software Co-Design. Procedings of the IEEE 85(3), 349–365 (1997)
23. Kastner, R., et al.: Instruction generation for hybrid reconfigurable systems. In: Proc. of the 2001 IEEE/ACM Intl. Conf. on Computer-Aided Design (ICCAD 2001). IEEE Press, San Jose (2001)
24. Atasu, K., Pozzi, L., Ienne, P.: Automatic application-specific instruction-set extensions under microarchitectural constraints. In: Proc. of the 40th ACM/IEEE Design Automation Conf. (DAC 2003). ACM Press, Anaheim (2003)
25. Nayak, A., et al.: Precision and Error Analysis of MATLAB Applications During Automated Hardware Synthesis for FPGAs. In: Design, Automation and Test Conf. in Europe (DATE 2001). IEEE Press, Munich (2001)
26. So, B., Hall, M., Ziegler, H.: Custom Data Layout for Memory Parallelism. In: Intl. Symp. on Code Generation and Optimization (CGO 2004). IEEE Computer Society Press, Palo Alto (2004)

27. Ziegler, H., Malusare, P., Diniz, P.: Array Replication to Increase Parallelism in Applications Mapped to Configurable Architectures. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) LCPC 2005. LNCS, vol. 4339, pp. 62–75. Springer, Heidelberg (2006)
28. Cardoso, J.M.P., Fernandes, J., Monteiro, M.: Adding Aspect-Oriented Features to MATLAB. In: SPLAT! 2006, Software Engineering Properties of Languages and Aspect Technologies, a Workshop Affiliated with AOSD 2006, Germany (2006)
29. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional, Reading (June 16, 2000)
30. Cordy, J.: The TXL Source Transformation Language. Science of Computer Prog. 61(3), 190–210 (2006)
31. Balland, E., et al.: Tom: Piggybacking rewriting on java. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 36–47. Springer, Heidelberg (2007)
32. Bravenboer, M., et al.: Stratego/XT 0.17. A language and toolset for program transformation. Science of Computer Prog. 72(1-2), 52–70 (2008)

Model Transformation Chains and Model Management for End-to-End Performance Decision Support

Mathias Fritzsche[1] and Wasif Gilani[2]

[1] SAP AG, Architecture and Innovation Services
Modeling and Taxonomy
Germany
mathias.fritzsche@sap.com
[2] SAP Research Belfast
Enterprise Intelligence
United Kingdom
wasif.gilani@sap.com

Abstract. The prototypical Model-Driven Performance Engineering (MDPE) Workbench from SAP Research permits multi-paradigm decision support for performance related questions in terms of what-if simulations, sensitivity analyses and optimizations. This support is beneficial if business analysts are designing new processes, modifying existing ones or optimizing processes. The functionality is provided as an extension of existing Process Modelling Tools, such as the tools employed by process environments like the jCOM! or the SAP NetWeaver Business Process Management (BPM) Suites as well as classical enterprise software like SAP Business Suite or Open ERP.

By evaluating our workbench for real world cases we experienced that business processes may span different environments, each employing different Process Modelling Tools. The presence of heterogeneous tools influences the end-to-end performance of the overall process. Thus, the MDPE Workbench essentially needs to take the complete process into account. In this paper, a model transformation chain and a model management architecture is explained to enable such functionality. This architecture combines results from our previous publications, outlines these results in more detail and explains them in the context of end-to-end processes. Furthermore, the work is evaluated with an industrial business process which spans three different Process Modelling Tools.

1 Introduction

We experienced that performance related decision support for business processes is especially useful in cases of a high degree of complexity in resource intensive processes, such as processes with layered use of resources or complex workflows. Also the complex statistical distribution of the history data or the plan data related to a process, like a planned workload, demand performance decision support. Business performance related decision support therefore needs to be

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 345–363, 2011.

considered as one integral part of process environments, especially of *Process Modelling Tools*. Such tools are considered as part of such process environments and are the focus of this paper.

In our previous publications we described the Model-Driven Performance Engineering (MDPE) Workbench [1,2,3] which enables the integration of multiple *Performance Analysis Tools* into Process Modelling Tools in order to enable support for decisions which have influence on the process performance. For instance, decision support related to throughput and utilization of resources can be provided by discrete event simulation tools, like AnyLogic [4]. Such tools also enable predictions related to the gross execution time of process instances. This enables to answer questions like "How will the execution time of a Sales Ordering Process be affected in June by adding an additional process step into the process in May?". Additionally, we did experiments with analytical Performance Analysis Tools, such as the LQN tool [5] and the FMC-QE tool [6,7]. Analytical tools are beneficial to answer sensitivity related questions in a short computation times, such as "Which are the most sensitive resources (humans or hardware) of the process for the overall performance?". Moreover, optimization tools can be integrated for performance related decision support, for instance, in order to decide at which point of time a certain business resource is needed to meet processing targets and to optimize utilization of resources. For optimization, existing libraries and tools can be used, such as OptQuest [8] which is employed by the AnyLogic tool as well.

Thus, the integration of different Performance Analysis Tools in Process Modelling Tools via the MDPE Workbench permits multi-paradigm decision support for process modelling. Additionally, we also experienced the need for decision support, which spans across multiple Process Modelling Tools. An example use case would be a business process provided by an enterprise software vendor, like SAP, which is extended with sub-processes. These can be maintained by independent software vendors. In such cases, different parts of the process might be modelled with different Process Modelling Tools which are normally based on different process modelling language, such as BPMN [9], jPASS! [10] or SAP proprietary languages. Thus, the performance related decision support needs to abstract these different languages.

In this paper, we describe the model transformation chain that we have implemented in order to integrate multi-paradigm decision support into multiple Process Modelling Tools. This transformation chain has raised the need for a model management architecture to support tracing of performance analysis results back to the process models, management of process model annotations and administration of the different tools.

The remainder of the paper is structured as follows: The next section describes an industrial example of a Process Modelling Tool chain. In Sections 3 and 4, a solution is proposed which comprises a model transformation chain and a model management approach. The success of this approach is evaluated in Section 5. Finally, in Sections 6 and 7, an overview on the related work and conclusions are provided.

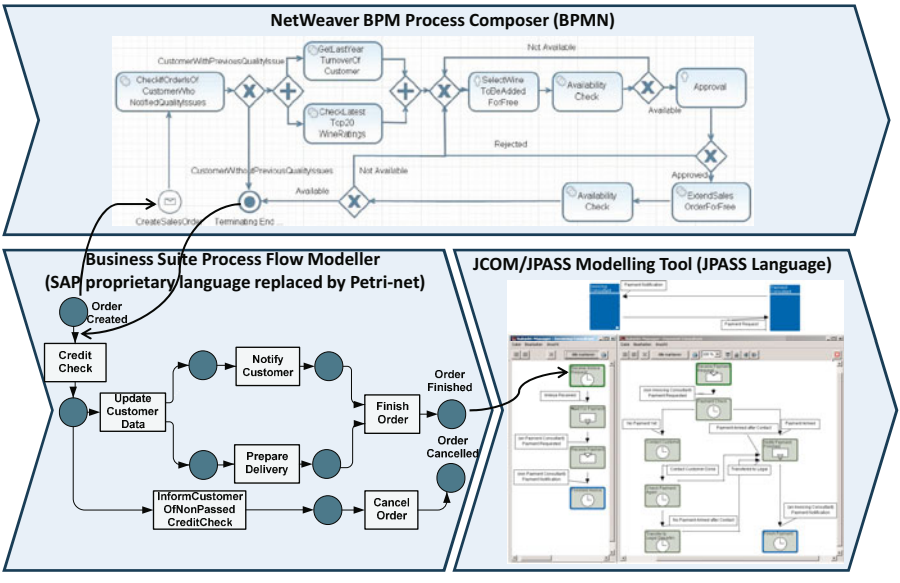


Fig. 1. Case Study: A business process spanning across three different Process Modelling Tools

2 Case Study: A Business Process Spanning Multiple Process Modelling Tools

Our case study involves a *Wine Seller* who gets wine supply from several suppliers, and thereafter sells the wine to customers. The *Sales and Distribution Organization* of the *Wine Seller* is supported by a standard software product implementing standard back-end processes, such as Business ByDesign [11] or Business Suite [12]. The back-end process under consideration is called "Sales Order Processing".

The lower left part of figure 1 depicts the "Sales Order Processing" process. Please note that the "Sales Order Processing" sub-process is modelled as so called "Process Flow Model". This is a SAP proprietary and, at the time of writing, not published modelling language. Therefore, the language has been replaced in the figure with a Petri-net [13].

Moreover, as can be seen in the lower right part of Figure 1, this process also triggers an external process for the "Invoice Processing", which is outsourced by the Wine Seller to an external company. In our case study, this external company uses a tool called *jPASS!*, which is the Process Modelling Tool of the jCOM! Business Process Management (BPM) Suite [14]. The jPASS! Process Modelling Tool enables top-down modelling of business processes with the help of the jPASS! modelling language [10,15].

The *Sales and Distribution Organization* of the Wine Seller additionally required an extension of the standard business process so that an extra free bottle of wine could be added to orders of customers who reported a quality issue in their previous order.

This raises the need for an extended version of the "Sales Order Processing" in this case. It is, however, not desirable to change the business process directly in the back-end because the application should be independent of the software vendors life cycle, SAP and jCOM! in our case. Therefore, a third technology is needed that could use the platform back-end business logic. To meet this requirement, the process called "Wine Under Special Treatment" has been developed, which is depicted by the upper part of Figure 1 and was originally introduced in [16].

For the implementation of this process, the BPMN [9] based Process Modelling Tool called "Process Composer", which is a part of the NetWeaver BPM [17] tooling, has been employed. This tool is specifically meant for applying extensions on top of existing back-end processes, such as the described extensions.

We experienced that a domain expert, with no performance expertise, generally cannot predict the performance related consequences if such an extension is deployed. In the context of the wine seller case study, one performance related issue would be to analyse if the additional manual approval steps introduced with the NetWeaver BPM result in an increased end-to-end time than defined as an objective and if so, how to improve the situation.

We also experienced that such analyses need to be done with a combination of different Performance Analysis Tools, for instance, a discrete event simulation tool for what-if questions, an analytical performance analysis tool for sensitivity related questions and an optimization tool to improve the process within user provided constraints.

Moreover, the complete end-to-end process spanning three different Process Modelling Tools needs to be considered since the whole process would be influenced by the newly developed composite process.

In the following sections, these needs are addressed.

3 A Model Transformation Chain for MDPE

In this section, the transformation chain of Figure 2 is proposed to interconnect chains of Process Modelling Tools with multiple Performance Analysis Tools.

This chain has a number of *Process Models* and *Process Model Annotations* as input. This input is transformed via a number of model to model transformations (see *M2Ms* in the Figure), a few model to text transformations (see *M2Ts* in the figure) and a number of intermediate models, into a *Performance Analysis Tool Input*. Most transformations create a trace model (see *Trace Models* in the Figure) that enable visualization of performance analysis results based on the original process models.

In the remainder of the section, the transformation chain is explained in more detail by describing the different intermediate models of this chain. This description is divided into two parts. The first part, which deals with the abstraction

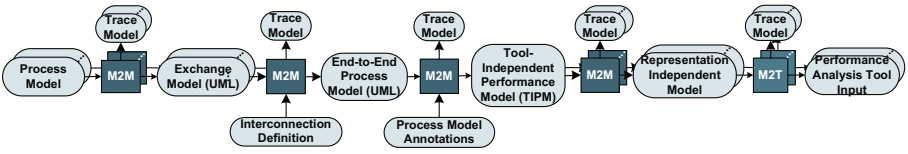


Fig. 2. MDPE transformation chain as block diagram [18]

of Performance Analysis Tools, is explained in the following paragraph. Then, the abstraction of Performance Modelling Tools is provided as the second part.

3.1 Abstraction of Performance Analysis Tools (TIPM and Representation Independent Model)

An abstraction of Performance Analysis Tools is reflected by the so called Tool-Independent Performance Model (TIPM) which we defined in a joint work together with TU-Dresden, XJ-Technologies. The TIPM is related to the so called Core Scenario Model [19]. A simplified version of the TIPM meta-model is shown in Figure 3. In the following paragraph, the meta-model is explained in detail.

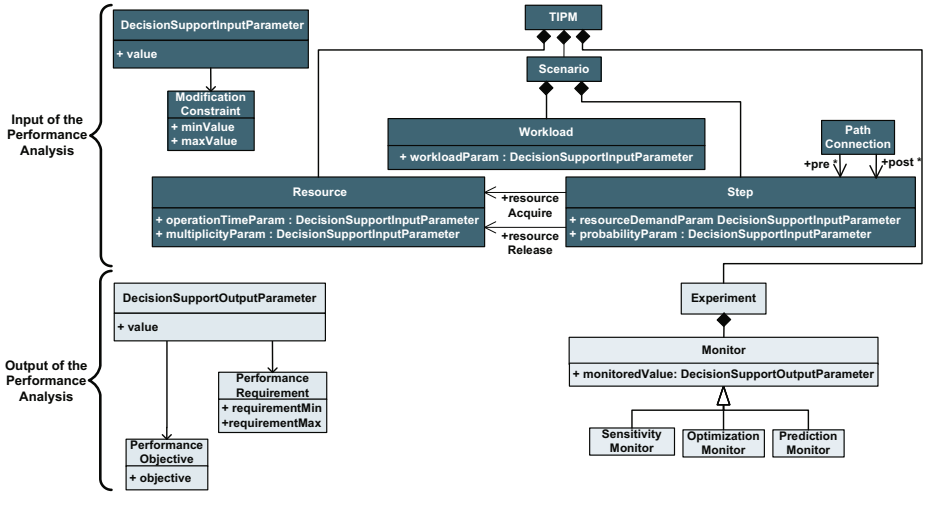


Fig. 3. Tool-Independent Performance Model (TIPM) (simplified)

A TIPM can be distinguished into two parts. The first part contains "Experiments" which include a list of "Monitors". These monitors are visualized with the light colour in Figure 3. Monitors are filled with values based on the performance analysis results, such as simulation results (see "PredictionMonitor" in the figure) which comprises latencies, utilizations and queue lengths. The latency specifies the time required in a simulation between two "Steps". The queue length and utilization attributes are filled with the queuing behaviour of the simulated

"Resources". The monitored values are of type "DecisionSupportOutputParameter" which can be associated with "PerformanceRequirements" and "PerformanceObjectives". These meta-classes represent user provided knowledge, to further analyse the performance analysis results, for instance, for the assessment if a requirement will be met in the future. User provided knowledge is taken from the *Process Model Annotations* (see Figure 2) which are explained in a later part of this section.

The part of the TIPM meta-model which is shaded in dark grey (see Figure 3) has to be filled before a performance analysis can be executed by a Performance Analysis Engine. This part of the TIPM combines the behavioural information from Process Models with Performance Parameters. The behavioural information, consumed from an *End-to-End UML Model* (see Figure 2) which is explained in the next paragraph, is defined in the TIPM as a graph with "Steps" (nodes) and "PathConnections", which are used to interconnect the "Steps".

Performance parameters, which are also taken from the process model annotations, are represented in the TIPM as attributes of different meta-classes. For instance, the parameter multiplicity (see "multiplicityParam" attribute in the "Resource" meta-class) indicates how many units are available in a pool of resources, e.g. 10 employees in the Philadelphia sales office.

Performance parameters are also used in order to create "Scenarios" in the TIPM. An example of such a "Scenario" is the execution of the previously introduced business process of the Wine Seller for a certain sales unit (e.g. *Philadelphia* and *Chicago*) or a certain type of process instances (e.g. *instances of sales orders below 1000 Euro* and *instances of orders of 1000 Euro or above*). Resources can be shared among multiple "Scenarios" (see Figure 3), such as the case that 10 employees for marketing are shared between the sales unit in Philadelphia and in Chicago. In this specific case, the TIPM would contain the business process of Section 2 twice, but the marketing resources only once. Of course, "Scenarios" can also be used to simulate resource sharing situations between different business processes.

All performance parameters are of type "DecisionSupportInputParameter". Such values can have a reference to a "ModificationConstraint", again based on process model annotations, which define possible variations of a performance parameter, for instance, in order to restrict the solution space for the optimization of a parameter to answer questions like "How many employees are needed to meet processing targets and to optimize utilization of resources?".

In the case that a Performance Analysis tool is not TIPM based and its input data structure is different to the TIPM structure, a transformation between the TIPM and the tool input is required. For such transformations, two concerns can be separated:

- Structural concern: One is required to perform a structural transformation from the TIPM structure to the Performance Analysis Tool structure. It is, for instance, necessary to generate a structure of AnyLogic library objects if the simulation tool AnyLogic [4] is employed as performance Analysis Tool. This structure includes all AnyLogic objects and their connections. This step,

therefore, needs to generate all necessary objects together with additional objects required to connect everything into a working model.
- Representation concern: The Performance Analysis Tools use a specific concrete syntax, such as a concrete XML format, as input. For the purpose of serialization it is necessary to apply the formatting so that the generated data can be read by the Performance Analysis Tool.

Thus, a so-called *Representation Independent Performance Analysis Model* (see Figure 2) is needed as an intermediate model to keep the transformations between a TIPM and Performance Analysis Tools clean from serialization details. Due to this intermediate models, two separate M2M transformation are introduced, one for the structural and another for the representation concern.

The structural model-to-model (M2M) transformation translates from the abstract syntax of the TIPM to the representation independent model. The second model-to-text (M2T) transformation then translates the representation independent model into the *Performance Analysis Input*. For example, if a XML based representation is used, this input would reflect a structure of XML Attributes and XML Nodes. Fortunately, most modern Performance Analysis Tools use a XML language to represent their input. One can therefore normally benefit from the already available serialization functionality of most transformation tools, such as the ATL tooling, to serialize XML conformant text.

In the following subsection, the intermediate models to abstract Process Modelling Tools are described.

3.2 Abstraction of Process Modelling Tools (Exchange Model and End-to-End Process Model)

Transformations from Process Models and Performance Parameters to a TIPM are complex as the structure of the TIPM meta-model is most likely different from the meta-model structure of the process modelling languages. This is due to the fact that Process Models normally express Petri-net [13] like behaviour whereas the TIPM additionally expresses "Scenarios", "Resources", etc., and the related associations.

We, however, also experienced that the structures of the process modelling languages, such as BPMN, jPASS!, Process Flow, etc., are related as they can normally be translated into each other.

Therefore, it is beneficial to add an *Exchange Model* in the Model Transformation Chain (see Figure 2) in order to express process behaviour. We have not chosen Petri-nets itself due to the following reasons: A UML to TIPM transformation was available from the initial proof of concept phase of the MDPE related research described in [3]. At that time, UML had especially been chosen as it enabled us share models with external partners in public funded research project, such as the MODELPLEX project [20]. Additionally, one can apply formally defined Petri-net semantics [13] to UML Activity Diagrams, as discussed in more detail in [21]. Finally, UML Activity diagrams permit to not only support active systems but, in future versions of the MDPE Workbench, also reactive systems as explained below.

Eshuis [22] analysed the process modelling related concept of workflows, which are used to define "operational business process[es]". According to Eshuis, models which do not consider interactions with the environment, such as timing events in BPMN, are interpreted as closed and active systems, whereas models which support external events are interpreted as open and reactive systems. He also explains that Petri-nets especially well support the first type, whereas UML Activity Diagrams also support the open and reactive systems. In our current implementation we only consider closed systems, i.e. we don't support most of the BPMN event types. By employing UML Activity Diagrams as exchange language, we are, however, prepared for implementing the support for the missing events in the future.

Due to the UML Activity Diagrams that we use as Exchange Models, the required transformations to adapt a new Process Modelling Tool are less complex than generating a TIPM directly. This is due to the fact that an complex already available UML to TIPM transformation can be reused. This also helps to prevent investing significant duplicated effort in implementing the similar functionality of transforming from Petri-net like behaviour model to a TIPM.

Figure 2 shows that we employ UML Activity Diagrams not only as Exchange Models, but also for an *End-to-End Process Model*. This model simply merges the different sub-processes into a single end-to-end process. The M2M transformation which generates this end-to-end model takes, in addition to the UML models for the sub-processes, also an *Interconnection Definition* as input. This definition is needed to specify how the different UML models are connected, for example, via a map which relates the final nodes of one sub-process with the initial nodes of another sub-process.

3.3 Challenges Based on the Model Transformation Chain

It has been explained that a transformation chain is needed to interconnect the different tools.

However, means are still needed to represent the transformation chain in a systematic way, so that, for example, different adapters for Process Modelling Tools and Performance Analysis Tools can be activated and deactivated with little effort and the modular transformation can still be extracted.

Related to this, it is necessary to associate trace models of a transformation chain instance with the correct intermediate models of the chain.

In addition to that, the process model annotations have to be managed by the model transformation chains. This is caused by the fact that the performance parameters and the user provided knowledge in terms of performance objectives, requirements and constraints is annotated based on the process models at the beginning of the chain, but only taken as an input for the transformation which generates the TIPM. This can be, for instance, the third transformation step in the transformation chain. Thus, an approach is required which makes it possible to keep annotations independent from a concrete intermediate model in a model transformation chain. Hence, the pollution of the proposed intermediate models and model transformations with the annotation data should be avoided.

Concluding, an approach is required which allows representation of different kinds of models, such as trace models, annotation models, intermediate models and also model transformations[1]. Additionally, for the purpose of navigation between the different models, an approach to manage relationships between the different models, such as trace relationships or annotation relationships, is necessary. In the following section, a solution for this need is provided.

4 A Model Management Approach for MDPE

In this section, first a meta-model is described which permits to define models and relationships between these models. Second, an architecture is proposed which employs this meta-model to simplify administration of the MDPE transformation chain, tracing and model annotation.

4.1 Representation of Models and Model Relationships

In order to systematically deal with different kinds of models and the relationships among them, Bézivin's megamodelling approach [23] enables to define all these relationships in another model. Such a global model could be interpreted for model navigation tasks, such as for the navigation from an intermediate model to the related trace models. Bézivin's basic meta-model for such a global model is depicted by the upper part of Figure 4. This meta-model enables to express different kinds of "Relationships" among different types of "Models", such as "Terminal Models". Thus, a number of process models and other kinds of models of the transformation chain are terminal models that can be described using the basic megamodelling concepts.

The upper right part of Figure 4 shows other types of terminal models, which are considered by the megamodelling approach, such as *Transformation Models* and *Weaving Models*.

However, besides of transformation models, annotation models and tracing models are also used in the transformation chain (see Figure 2). Even if annotation models and tracing models are based on weaving models, a distinguishing between both types is necessary to simplify end-to-end tracing and transformation chain based annotation functionality.

The "ModelAnnotation" and "ModelTrace" relationships have been introduced to allow navigation among models related through annotation and traceability information. The last one is associated with a "TraceModel" which can be related to a specific target model through associations inherited-from "DirectedRelationship" and "ModelWeaving". In a similar way, "ModelAnnotation" is associated with an "AnnotationModel" and can be related to a specific target model. The "ModelTransformation" meta-class in Figure 4 represents a single transformation. A "ModelTransformation" can be specified in terms of a "TransformationModel", such as an ATL transformation.

[1] In this paper, we assume that transformation scripts conform-to meta-models, such as it is the case for ATL scripts.

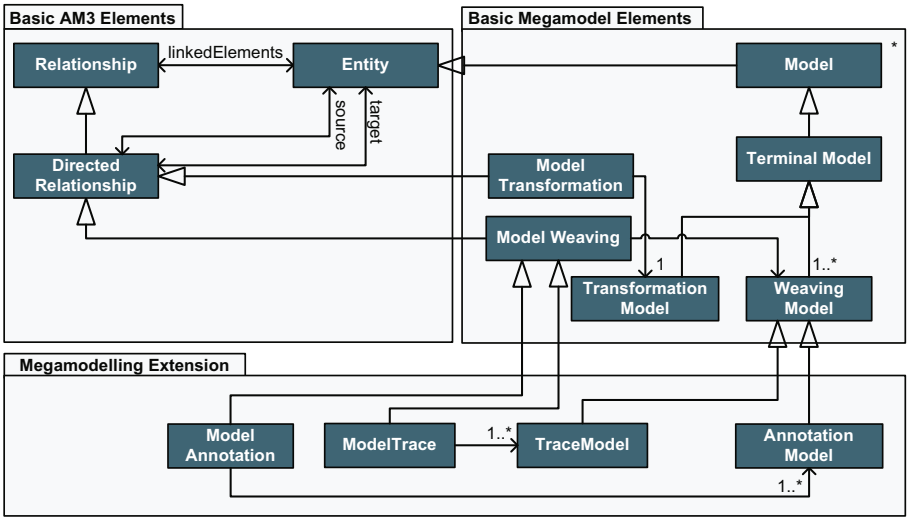


Fig. 4. Extract of the MDPE Metamodel extension of the Megamodel

The following section explains how the megamodel is employed within the MDPE Workbench [2,24,25].

4.2 Magamodelling Based Model Navigation

Based on the refined megamodel, the architecture of the *Model Navigation Agent* of the MDPE Workbench has been defined. This architecture is depicted in Figure 5.

The figure shows that, beside of other relationships, the *ModelTransformation (Relationship)*, *ModelTrace(Relationship)* and *ModelAnnotation(Relationship)* are represented in the *Refined Megamodel*. These relationships and the different *MDPE Modelling Artefacts* are used by the *Transformation Chain Controller*, *Tracing Controller* and *Annotation Controller* as depicted in the figure.

In the following paragraphs, the different controllers are explained in detail.

Transformation Controller: Figure 6 shows that the transformation relationship is set in the megamodel by an *Administration Tool*. The purpose of this tool is to enable users to select the currently active Process Modelling Tool and Performance Analysis Tool. The administration tool therefore stores the currently active Transformation Chain into the megamodel. This specification is consumed by a *Transformation Controller*, which executes the different *Transformation Models* and outputs the final Tool Input for Performance Analysis Tools based on a number of *Process Models*. The transformation outputs a number of *Trace Models* as by-products of the transformation [25]. The Transformation Controller also sets the *Tracing Relationships* between Process Models, Intermediate Models (e.g. the TIPM) and the final input for the Performance Analysis Tool into the megamodel.

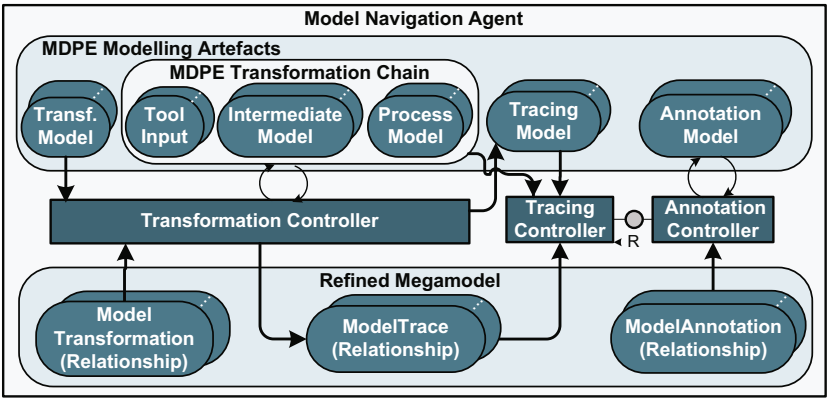


Fig. 5. Model Navigation Agent

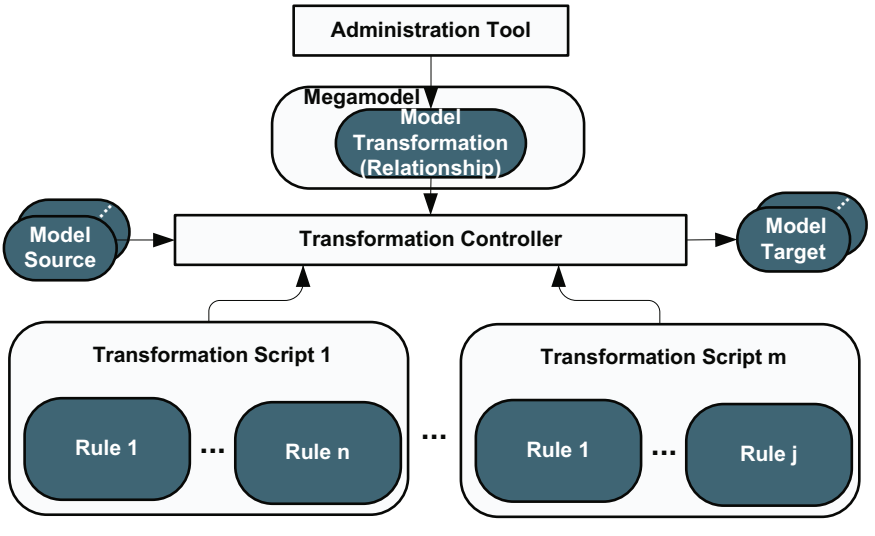


Fig. 6. Megamodel based Transformation Controller

Tracing Controller: The tracing relationships between trace models and models of the transformation chain are consumed by the *Tracing Controller*, which is depicted in Figure 7. Due to the *Model Trace* relationships in the megamodel, this agent is able to navigate backward through an instance of a model transformation chain containing i sets of intermediate, source and target models as shown in the figure. The agent is therefore also able to create an end-to-end trace model from an arbitrary Set_h of models to a Set_k of models. The resulting trace model is called the *End-to-End Trace Model* in the figure. The implementation of this end-to-end trace model generation has been done with the ATL transformation language.

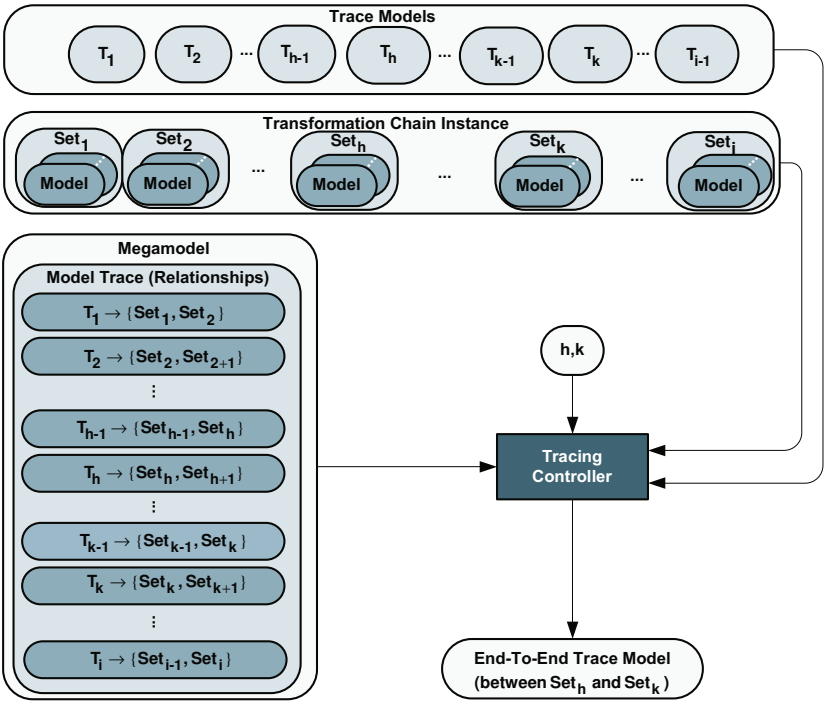


Fig. 7. Megamodel based Tracing Controller

However, some M2M languages, such as ATL, only allow transformation scripts where each single input model is explicitly named. Thus, lists of an infinite length of input models are not always supported. This is, however, beneficial for tools like the MDPE Workbench if one considers possible future extensions of the approach. Additionally, it is only required to activate the "UML2UML Transformation" if a process is spanning across multiple process environments. Thus, only some set-ups of the MDPE Workbench will contain this transformation. Therefore, an end-to-end trace model can be generated via recursively iterating over the on the traces from the different transformation steps, for

instance, via a transformation which takes all traces and the megamodel as an input. The megamodel is needed because the transformation needs to know the currently active model transformation chain.

The resulting end-to-end trace model can be utilized for different purposes, for instance, to utilize a model annotation as input for an arbitrary transformation step in an automated model transformation chain (see "R" between Tracing Controller and Annotation Controller in Figure 5), as described in the following subsection.

Annotation Controller: Figure 8 gives an overview of the proposed *Annotation Controller*. This controller takes the end-to-end trace model from the tracing controller as input in order to create an *Annotation of Model in* Set_h based on *Data of Model in* Set_k.

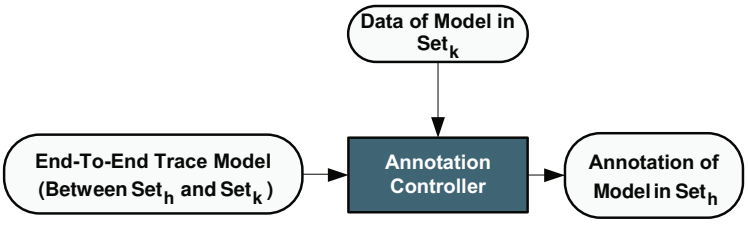


Fig. 8. Megamodel based Annotation Controller

The approach can, therefore, be used to annotate content or annotations of an arbitrary model in the Set_k in an automated transformation chain, to another arbitrary model in the Set_h. Note that the Set_h is not necessarily created in the chain before the Set_k.

The combination of the tracing controller and the annotation controller is, however, only beneficial if the following restriction is considered:

In some cases a model in the Set_k cannot be related to a model in the Set_h. This for example happens if not all meta-classes of a model in the Set_k are translated into models of Set_h. In this case it is obviously not possible to relate these elements of the Set_k to elements of the Set_h, which have not been translated. However, such cases are considered as an indication of either an invalid model annotation or an invalid model transformation chain. Thus, the domain specialist needs to be informed by the tooling in such cases.

5 Experiences

For the evaluation of the transformation chain based approach, we had to set up the model transformation chain as shown in Figure 9. This transformation chain takes the three different kinds of process modelling languages as input, which constitute the end-to-end process described in Section 2. All three modelling languages are first transformed to UML (see *BPMN2UML*, *ProcessFlow2UML*,

JPASS2UML in Figure 9). In a second step, the three UML models are interconnected to formulate an end-to-end UML model (see *UML2TIPM* in Figure 9), which is then, together with the Performance Parameters, transformed to a TIPM.

Three different types of performance related decision support are provided. The AnyLogic tool is used as a discrete event simulation engine and as an optimization engine. Please note that the required transformations are different for both cases as different kinds of experiments have to be created in the AnyLogic tool. Moreover, the simulation based decision support only takes performance requirements into account as Performance Assessment Data, whereas the optimization also considers objectives and constraints. Additionally, we attached the FMC-QE tool in order to get analytic support for sensitivity related decisions. Therefore, the TIPM, which contains the end-to-end business process including its resource related behaviour, is transformed via a Representation Independent Model (e.g. the *AL_SIM Model* in Figure 9) into Tool Input (e.g. the *AL_SIM.xml* in Figure 9).

In the following paragraphs, we describe our experiences:

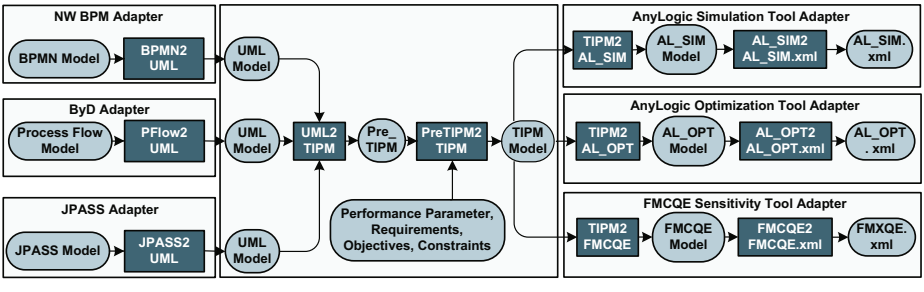


Fig. 9. Example Transformation Chain Set-Up

From the functional point of view, the combination of TIPM and UML as intermediate languages enables to abstract the different process modelling languages. Without the proposed transformation chain, the creation of end-to-end simulation models would have required to manually switch between three different Process Modelling Tools and three different Performance Analysis Engines. Therefore, in case one wants to get end-to-end decision support for the use case that we introduced, nine different tools need to be understood. In general, for the end-to-end decision support spanning across n process environments and m Performance Analysis Tools, $n+m$ tools have to be understood by a domain specialist. In [26] metrics are provided to measure the complexity of using a tool. One of the measures is the number of manual steps; another one is the number of context switches. Obviously, our solution reduces the number of manual steps to a "push-button" activity and one is not required to switch between numerous tools involved.

Due to UML being employed as an Exchange Model, we only had to write the complex part of the transformation between process modelling languages and TIPM once. Thus, as long as a new process modelling language can be mapped to UML Activity Diagrams, those effectively reduce the development effort associated with attaching different Process Modelling Tools. If a process modelling language cannot be mapped to UML Activity Diagrams due to a currently unforeseen process modelling concept, one might still be able to map it, with more effort, to the TIPM. If this is also not possible, significant development effort might be required to modify the TIPM and, in worst case, all transformations from and to the TIPM. This is, however, considered as not very likely.

Additionally, the TIPM as Generic Performance Analysis Model also enabled to provide decision support based on multiple Performance Analysis Engines, such as the analytic FMC-QE tool and the simulation engine AnyLogic. In case a new Process Modelling Tool is attached to the MDPE Workbench, the existing transformations between TIPM and Performance Analysis Tools can be reused. Again, if the TIPM needs to be modified for a new Performance Analysis Tool due to a currently unforeseen performance analysis concept, significant effort might be needed, in particular if the new TIPM version induces the need to update existing transformations. This is, however, considered as not likely.

Moreover, the Transformation Chain Management permits to manage the different modelling artefacts, such as annotation models and process models, across the transformation chain. This enables one to use, for instance, annotation models that reference the process models at any step in the transformation chain, and by tracing Assessment Results backwards through the chain in order to visualize them, also as annotation models, based on the original process models.

Adding a new step to the Transformation Chain is, therefore, not an issue anymore. This was, for instance, needed when we added the "UML2UML" transformation step (see Figure 9). The additional transformation step was one prerequisite in order to realize end-to-end decision support which spans the tool chain built by the Process Flow modelling tool, the Process Composer and the jPASS! tool.

However, the performance parameters are currently specified manually, which is not sufficient in cases where historic performance parameters, such as how long a special process step took in the past, are available as process instance data. Therefore, in order to fully support end-to-end decision support spanning across the three BPM tools, it is not only required to abstract three different modelling languages with a model transformation chain, but also to abstract three different sources for such historic process instance data.

Additionally, we are lacking a mechanism to deal with the confidentiality of the business sub-processes. It might, for instance, not be appropriate for the third party of our use case to provide their business process description to the wine seller company. Thus, solutions are required to ensure confidentiality, for instance, by preventing that the employees of the wine company have visibility to the invoicing process.

6 Related Work

Some model-driven scenarios which involve chains of multiple transformations can be found in the literature, such as the twelve-step transformation chain in the interoperability scenario for business rules presented in [27] or the five step chain in the interoperability scenario for code clone tools presented in [28]. Compared to these works, reasoning for the transformation chain of the MDPE Workbench has been provided to integrate chains of Process Modelling Tools and Performance Analysis Tools. Additionally, an architecture for model management has been provided which enables end-to-end tracing, annotation and administration of automated model transformation chains of arbitrary lengths.

The main concepts behind the concept of intermediate languages that we have employed for our transformation chain can also be found in other software transformations. For example, the abstraction of Performance Analysis Tools and Process Modelling Tools with performance specific language TIPM and the behaviour specific language UML Activity Diagrams is related to formats like PDG or SSA [29,30]. The combination of UML Activity Diagrams and the TIPM can be seen as means for a source- or target-independent intermediate (pivot) format to reduce the number of required transformations. This is similar to the use of byte code to abstract from the processor architecture.

From the application point of view, the closest related work to our knowledge is the integrated performance simulation functionality within some process environments of the BPM domain, such as EMC Documentum Process Suite [31]. However, the offered simulation-based functionality is only at a basic level [32]. To the best of our knowledge, the proposed approach is the only one which provides, based on a model transformation chain and an architecture for model management, support for chains of process environments, each employing different Process Modelling Tools. Our approach further enables to benefit from the know-how and functionality contained in a sophisticated performance decision support system, which makes use of sophisticated model simulations, optimizations, and static analyses, and also a combination of them.

7 Conclusion

In this paper, the combination of a model transformation chain and a model management architecture has been proposed in order to provide end-to-end performance related decision support for business processes spanning multiple Process Modelling Tools. The proposed architecture enables to interconnect multiple Process Modelling Tools with multiple Performance Analysis Engines.

As a next step we anticipate to extend the current model transformation chain in order to provide functionality for the abstraction of history data logs which are provided by process environments. This would allow a better integration of the MDPE Workbench into these environments. Additionally, we plan to extend the existing model management approach to permit dealing with the confidentiality of the business sub-processes.

Disclaimer

References

1. Fritzsche, M., Picht, M., Gilani, W., Spence, I., Brown, J., Kilpatrick, P.: Extending BPM Environments of your choice with Performance related Decision Support. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 97–112. Springer, Heidelberg (2009)
2. Fritzsche, M., Johannes, J., Assmann, U., Mitschke, S., Gilani, W., Spence, I., Brown, J., Kilpatrick, P.: Systematic usage of embedded modelling languages in automated model transformation chains. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 134–150. Springer, Heidelberg (2009)

3. Fritzsche, M., Johannes, J.: Putting Performance Engineering into Model-Driven Engineering: Model-Driven Performance Engineering. In: Giese, H. (ed.) MODELS 2007. LNCS, vol. 5002, pp. 164–175. Springer, Heidelberg (2008)
4. XJ Technologies: AnyLogic — multi-paradigm simulation software (2009), http://www.xjtek.com/anylogic/
5. Franks, R.G.: PhD Thesis: Performance Analysis of Distributed Server Systems. Carlton University (1999)
6. Zorn, W.: FMC-QE: A new approach in quantitative modeling. In: Proceedings of the 2007 International Conference on Modeling, Simulation & Visualization Methods (MSV 2007), pp. 280–287. CSREA Press (2007)
7. Porzucek, T., Kluth, S., Fritzsche, M., Redlich, D.: Combination of a discrete event simulation and an analytical performance analysis through model-transformations. In: Proceedings of the International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2010), pp. 183–192. IEEE Computer Society, Los Alamitos (2010)
8. Rogers, P.: Optimum-seeking simulation in the design and control of manufacturing systems: Experience with optquest for arena. In: Proceedings of the 2002 Winter Simulation Conference, WSC 2002 (2002)
9. Object Management Group: Business Process Modeling Notation Specification, Final Adopted Specification, Version 1.0 (2006), http://www.bpmn.org/Documents/OMG%20Final%20Adopted%20BPMN%201-0%20Spec%2006-02-01.pdf
10. jCOM1 AG: jpass! - subjektorientierte prozessmodellierung (2009), http://www.jcom1.com/cms/jpass.html
11. SAP AG: Sap solutions for small businesses and midsize companies (2009), http://www.sap.com/solutions/sme/businessbydesign/index.epx
12. SAP AG: Sap business suite - integrated enterprise applications help lower costs, improve insight, and capture opportunities (2009), http://www.sap.com/solutions/business-suite/index.epx
13. Peterson, J.L.: Petri Net Theory and the Modelling of Systems. Prentice-Hall, Englewood Cliffs (1981)
14. jCOM1 AG: Process management - jcom1 (2009), http://www.jcom1.com/
15. Fleischmann, A.: Distributed Systems, Software Design & Implementation (1995)
16. Fritzsche, M., Gilani, W., Fritzsche, C., Spence, I., Kilpatrick, P., Brown, T.J.: Towards utilizing model-driven engineering of composite applications for business performance analysis. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 369–380. Springer, Heidelberg (2008)
17. SAP AG: Components & tools of sap netweaver (2009), http://www.sap.com/platform/netweaver/components/sapnetweaverbpm/index.epx
18. Knöpfel, A., Gröne, B., Tabeling, P.: Fundamental Modeling Concepts: Effective Communication of IT Systems. John Wiley & Sons, Chichester (2006)
19. amd Murray Woodside, D.B.P.: An intermediate metamodel with scenarios and resources for generating performance models from uml designs. Software and Systems Modeling 6(2), 163–184 (2007)
20. Information Society Technologies: Sixth Framework Programme, Description of Work: MODELling solution for comPLEX software systems (MODELPLEX) (2006)
21. Dehnert, J.: PhD Thesis: A Methodology for Workflow Modeling: From business process modeling towards sound workflow specification. TU-Berlin (2003)
22. Eshuis, R.: PhD Thesis: Semantics and Verification of UML Activity Diagrams for Workflow Modelling. Centre for Telematics and Information Technology (CTIT), University of Twente (2002)

23. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the large and modeling in the small. In: Aßmann, U., Liu, Y., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 33–46. Springer, Heidelberg (2005)
24. Fritzsche, M., Bruneliere, H., Vanhoof, B., Berbers, Y., Jouault, F., Gilani, W.: Applying megamodelling to model driven performance engineering. In: Proceedings of the International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2009), pp. 244–253. IEEE Computer Society, Los Alamitos (2009)
25. Fritzsche, M., Johannes, J., Zschaler, S., Zherebtsov, A., Terekhov, A.: Application of tracing techniques in model-driven performance engineering. In: Proceedings of the 4th ECMDA Traceability Workshop (ECMDA-TW), pp. 111–120 (2008)
26. Keller, A., Brown, A.B., Hellerstein, J.L.: A configuration complexity model and its application to a change management system. IEEE Computer Society Transactions 4, 13–27 (2007)
27. Fabro, M.D.D., Albert, P., Bzivin, J., Jouault, F.: Industrial-strength rule interoperability using model driven engineering. In: Proceedings of the 5mes Journes sur l'Inginierie Dirige par les Modles (2009) (to appear)
28. Sun, Y., Demirezen, Z., Jouault, F., Tairas, R., Gray, J.: A model engineering approach to tool interoperability. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 178–187. Springer, Heidelberg (2009)
29. Ferrante, J., Ottenstein, K., Warren, J.: The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS) 9, 319–349 (1987)
30. Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, F.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems (TOPLAS) 13, 451–490 (1991)
31. Associates, B.S.: The BPMS Report: EMC Documentum Process Suite 6.0 (2008), http://www.bpminstitute.org/whitepapers/whitepaper/article/emc-documentum-process-suite-6-0-1/news-browse/1.html
32. Harmon, P., Wolf, C.: The state of business process management (2008), http://www.bptrends.com/surveys_landing.cfm

Building Code Generators with Genesys: A Tutorial Introduction

Sven Jörges[1], Bernhard Steffen[1], and Tiziana Margaria[2]

[1] Chair of Programming Systems, Technische Universität Dortmund, Germany
sven.joerges@tu-dortmund.de, bernhard.steffen@cs.tu-dortmund.de
[2] Chair of Service and Software Engineering, Universität Potsdam, Germany
margaria@cs.uni-potsdam.de

Abstract. Automatic code generation is a key feature of model-driven approaches to software engineering. In previous publications on this topic, we showed that constructing code generators in a model-driven way provides a lot of advantages. We presented Genesys, a code generation framework which supports the model-driven construction of code generators based on service-oriented principles. With this methodology, concepts like bootstrapping and reuse of existing components enable a fast evolution of the code generation library. Furthermore, the robustness of the code generators profits from the application of formal methods. In this paper, we will show in detail how code generators are constructed with Genesys, in a tutorial-like fashion. As an example, we will build a code generator for HTML documentation from scratch.

1 Introduction

Automatic code generation is a key feature of model-driven approaches to software engineering. It has several advantages such as the elimination of manual coding errors and the avoidance of repetitive work, and it provides a fast track to a deployable and testable system/application.

In [1], we showed that constructing code generators in a model-driven way provides a lot of advantages. We presented Genesys, a code generation framework which supports the model-driven construction of code generators based on service-oriented principles. Genesys is an integral part of jABC [2–4], a flexible framework designed to enable systematic development according to the XMDD paradigm (cf. Sect. 2). When modeling with jABC, systems or applications are assembled on the basis of an (extensible) library of well-defined, reusable building blocks. From such models, code for various platforms (e.g. Java-based platforms, mobile devices, etc.) can be generated. As the code generators for these models have also been built within jABC, concepts like bootstrapping and reuse of existing components enable a fast evolution of the code generation library. Furthermore, the robustness of the code generators profits from the application of formal methods like e.g. model checking [5].

In this paper, we will show in detail how code generators are constructed with Genesys, in a tutorial-like fashion. As an example, we will build a code generator

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 364–385, 2011.

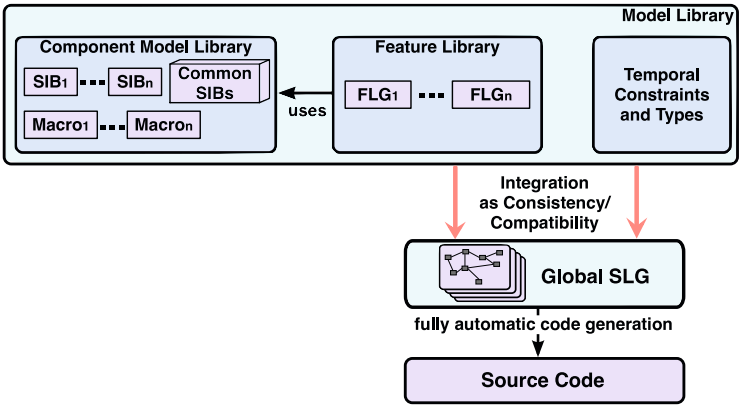


Fig. 1. How XMDD is realized by the jABC framework

for HTML documentation from scratch, thereby elaborating on the roles in the process and their respective tasks.

In the following sections, we first will briefly introduce the jABC framework and its basic concepts, as well as Genesys itself (Sect. 2 and 3). Afterwards we focus on the paper's main contribution, which is a detailed tutorial on how to build a code generator, the *Documentation Generator*, with Genesys (Sect. 4). In Sect. 5 we discuss related work. Finally, we will sum up and describe some future work in Sect. 6.

2 The jABC Framework

jABC [2–4] is a flexible framework designed to support systematic development according to the XMDD (Extreme Model-Driven Development) paradigm [6]. The key idea of XMDD is to move all application-specific activities to the modeling level. Models are put at the center of the design activity, becoming *the* first class citizens of the *global* system design process. Consequently, libraries are established at the model level, i.e. that building blocks are (elementary) models rather than software components, and systems are specified by model combinations (composition, configuration, superposition, conjunction etc.), viewed as a set of constraints that the implementation needs to satisfy. Via code generation, such model combinations are translated into a homogeneous solution for a desired environment. As opposed to other model-driven approaches such as MDA [7], generated code is regarded as a "by-product" that must never be touched manually. Any system changes (upgrades, customer-specific adaptations, new versions etc.) with visible effect on the models happen only at the modeling level, with a subsequent regeneration of the code.

Fig. 1 shows how the XMDD paradigm is realized by jABC. The box in the center depicts the central development artifact labeled "Global SLG", which represents an application built in jABC. SLG is short for *Service Logic Graph*,

which is the term for any model built with jABC. Basically, SLGs are directed graphs that represent the flow of actions in an application. The *model library* (big box on top of Fig. 1) provides the repertoire for modeling such SLGs. It is divided into three parts: the component model library, the feature model library, and a library of temporal constraints and types.

The *component model library* contains elementary, reusable building blocks required to assemble an application. Such a building block may represent an atomic service, providing a single functionality of e.g. a legacy system, COTS software or a web service. In jABC, these atomic services are called *Service Independent Building Blocks* (SIBs). jABC already ships with a huge library of such SIBs (the *Common SIBs* [8]), which provide a very elementary basis for assembling SLGs. The bulk of SIBs we use for the tutorial in this paper can be found among the *Common SIBs*. Besides being an atomic service, a building block may also represent a whole model (i.e. another SLG). Such building blocks are called *macros*. Thus SLGs can be hierarchical, which grants a high degree of reusability not only of the building blocks, but also of the models themselves, within larger systems (we will also demonstrate the reuse of models in this tutorial). Note that the only difference between a SIB and a macro is whether the component is realized by a concrete implementation or by a model - for the jABC user, a macro is used just like any other SIB.

A model that realizes a macro is called a "feature", which is reflected in Fig. 1 by the *feature library*. "Features" denote entire SLGs, modeling reusable application aspects such as error handling, security management, logging etc. Such aspects (called *Feature Logic Graphs*, FLGs) are modeled once, and afterwards they are part of the model library and can be reused across applications, even across domains.

Furthermore, the model library also includes *constraints*, which define the rules of an application and can be verified automatically by means of formal methods like model checking. Such constraints, combined with corresponding verification tools, assure the consistency and compatibility when the various parts of the model library are used to build the global application model (Fig. 1: "Integration as Consistency/Compatibility"). Having assured that this model resembles a valid application according to the given constraints especially clears the way for generating executable code for a desired target platform. This is the task of Genesys, which will be explained in Sect. 3.

The jABC framework provides a tool for graphically modeling SLGs from SIBs, macros and FLGs. Fig. 2 shows jABC's graphical user interface, which consists of three main parts (indicated by the numbers):

1. the *project and SIB browser*, which enable the user to browse available jABC projects and the library of building blocks that can be used for modeling,
2. the *graph canvas*, which is used for composing models, and
3. the *inspectors*, which provide detailed information about elements selected in the canvas.

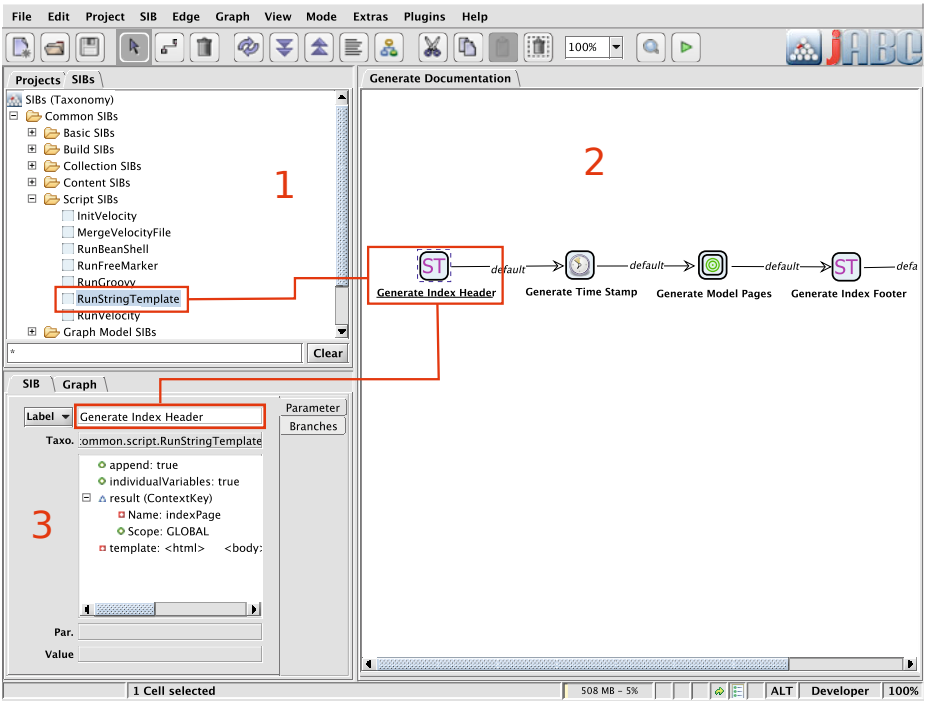


Fig. 2. jABC's User Interface

We will elaborate on this figure in the following sections, which further explain the concepts of SIBs and SLGs in order to provide a basis for the tutorial in this paper.

2.1 Service Independent Building Blocks (SIBs)

SIBs are the elementary building blocks for constructing models in jABC. As pointed out above, a SIB represents an atomic service, which can be everything, ranging from low-level functionality like string concatenation, displaying a message or reading information from a database, to web services or even the interaction with highly complex systems, like enterprise resource planning (ERP) software. For the user of jABC, the granularity of the service represented by such a SIB is completely transparent: In order to use a SIB, it is only necessary to know which behavior it represents, but not how the behavior is implemented.

These perspectives lead to two separate roles involved in the development with jABC. First of all, *application experts* use jABC to graphically build models of an application. They usually have very specific knowledge of this application and the relevant concepts of the underlying domain, but they do not care about concrete implementations or target platform issues (e.g. which database is used etc.). Application experts resort to ready-made libraries of SIBs and use these SIBs as simple black boxes, independent of their concrete realization. Contrary

to this, *SIB experts* (or *IT experts*) are developers who realize a SIB's behavior for a concrete target platform, i.e. they provide the concrete implementations of the atomic services. In practice, application experts and SIB experts work together closely: If the application expert needs a particular SIB that is not provided by a ready-made library, he requests a corresponding implementation from the SIB expert. In the following, we will elaborate on the SIB concepts from both perspectives.

The Application Expert's View on SIBs: As pointed out above, the application expert builds models by using SIBs from ready-made libraries, which are shipped with jABC. These libraries can be searched and explored using jABC's SIB browser[1], depicted in Fig. 2 (1). Via drag and drop, a particular SIB can be moved from the SIB browser to the canvas (2), which is technically tantamount to instantiating the SIB. Consequently, there can be several *instances* of one SIB in a model.

Fig. 2 (2) shows an example of such a SIB instance, highlighted by the box. In the SIB browser (1), it is visible that it is an instance of the SIB `RunStringTemplate`, which is categorized as part of the Common SIBs (cp. Sect. 2) package called "Script SIBs". The task of `RunStringTemplate` is to employ the template engine StringTemplate [9] to evaluate a template. Such a template is basically a textual skeleton containing placeholders which are filled with dynamic content as soon as the template engine is invoked.The corresponding SIB instance in the canvas is labelled `Generate Index Header`, and there is another instance of the SIB contained in the model (labelled `Generate Index Footer`), which illustrates the reusability of those building blocks. In the canvas, SIBs are visualized by an *icon* and a *label* at the bottom of the icon, both of them are freely customizable by the application expert. By selecting a particular SIB instance in the canvas, its details are displayed by the SIB inspector (3).

In order to facilitate reusability, each SIB provides a set of *parameters* for configuring the SIB's behavior. As visible from the SIB inspector (3), the SIB `RunStringTemplate` takes four parameters, e.g. one of them ("template") being the template that should be evaluated by StringTemplate. In order to allow SIB instances in a model to communicate with each other, i.e. to share data, the concrete service implementations usually keep track of an *execution context*. Technically, this context is like a hash map, containing simple key-value pairs. Thus a SIB instance is able to read and manipulate data that has been stored in the context by other SIB instances, provided that both SIB instances agree on the key which identifies the data. While the concrete implementation of the execution context is irrelevant for (and invisible to) the application expert, the keys used by SIB instances to share data are again customized by SIB parameters. Accordingly, the exemplary SIB instance in Fig. 2 has one parameter "result" (3), which specifies the key used to store the evaluation result of StringTemplate in the execution context (in the example, this key is `indexPage`). Besides parameters, each SIB also provides a set of so-called *branches*, which reflect its possible

[1] The categorization structure visualized there is a taxonomy [2].

execution results. For instance, the SIB `RunStringTemplate` has two branches (not visible in Fig. 2): *default*, if the template was evaluated successfully, and *error*, if the template could not be evaluated (e.g. because of syntax errors). As will be described in the following Sect. 2.2, branches provide the basis for wiring SIB instances in a model via directed edges.

The SIB Expert's View on SIBs: The SIB expert provides the application expert with required SIBs, either by continuously extending the ready-made libraries, or as a reaction to a direct request. Implementing a SIB basically consists of two parts: the SIB itself and its service adapters.

The *SIB* is the building block presented to the application expert in jABC. As pointed out above, this is a black box representing a specific behavior configurable by parameters, and reflecting its possible results by branches. Such a SIB is described by means of a very simple Java class, which defines the SIB's constituents via programming conventions:

- parameters are defined by all public fields of the Java class,
- branches are defined by a static String array called `BRANCHES`, and
- a default icon and documentation are added by implementation of special methods.

Finally, the class is marked as a SIB via an annotation (`@SIBClass`), which also declares a *unique identifier* in order to reliably distinguish the SIB from other SIBs. Sect. 4.2 shows a detailed example of such a Java class.

A *service adapter* implements the SIB's behavior for a concrete target platform. Particularly, as one SIB may be executable on multiple target platforms, an arbitrary number of service adapters can be attached to a SIB. A service adapter is usually implemented in a programming language supported by the desired target platform, so it may for instance be a Java class, a C# class or a Python script. Decoupling the concrete platform-specific implementations from the SIB description assures that the SIB itself is entirely platform-independent. As soon as at least one service adapter is implemented for a SIB, the SIB is *executable*, thus enabling:

- *Interpretation:* Models containing SIBs are directly executable by an interpreter which calls the implementation provided by one of the service adapters.
- *Code Generation:* When transforming models to code, the code generator translates each SIB instance to calls to the corresponding service adapter.

Sect. 4.2 shows an example of a service adapter.

2.2 Service Logic Graphs (SLGs)

As mentioned above, SLG is the term for any model built with jABC. Basically, SLGs are directed graphs that represent the flow of actions in an application. The nodes in such a graph are SIB instances or, in order to facilitate hierarchical modeling, macros that point to other SLGs. For instance, in the example

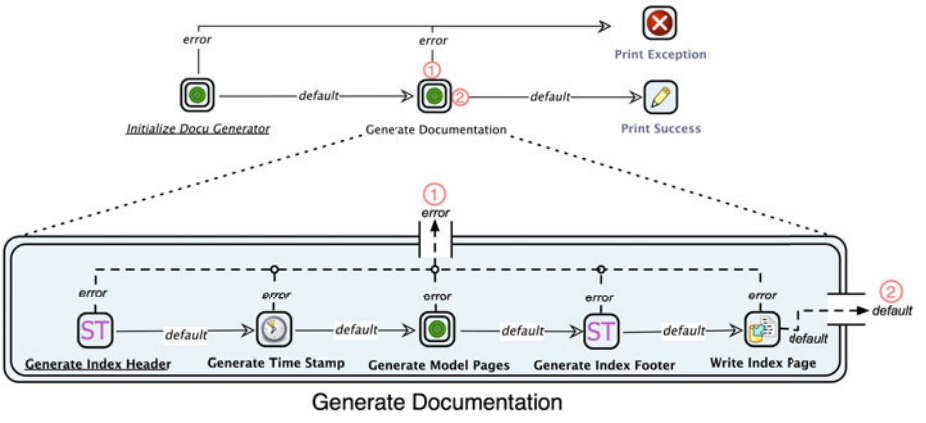


Fig. 3. Hierarchical Models (SLGs) in jABC

model depicted on the top of Fig. 3, the nodes labelled `Print Exception` and `Print Success` are SIB instances, while the nodes labelled `Initialize Docu Generator` and `Generate Documentation` are macros (indicated by the green dot on their icons).

The directed edges between the nodes indicate the flow of actions. Each edge is labelled with one or more branches, whereas the source node of the edge defines the set of possible branches that can be assigned to that edge. In other words, the wiring of SIB instances and macros in models is performed on the basis of possible execution results. Roughly speaking, branches could also be seen as "exits" of a SIB/macro. If a node has more than one outgoing edge, the edges represent alternative execution flows. For instance, the node `Initialize Docu Generator` in Fig. 3 has two branches "default" and "error", each assigned to one outgoing edge in the model. This reads as follows: *If the result of* `Initialize Docu Generator` *is "default" proceed with* `Generate Documentation`*, if the result is "error" go to* `Print Exception`*.* If `Initialize Docu Generator` produces another result, it is considered an undefined behavior. In order to define where the execution of a model starts, a node can be defined as an entry point. This is indicated by the node's label being underlined (e.g. `Initialize Docu Generator` in Fig. 3)[2].

The preceding descriptions already anticipated a mechanism which is key to seamlessly enabling hierarchical modeling: Just like SIBs, macros also have parameters and branches. However, as these parameters and branches belong to an entire model associated with the macro, they are called *model parameters* and *model branches*, respectively.

A model's set of model parameters and model branches is defined by selectively *exporting* parameters and branches of SIB instances/macros in that model. For model branches, this is exemplified in Fig. 3. The bottom part of the figure shows

[2] Please note that a model could potentially have more than one entry point, depending on whether this is supported by the selected interpreter or code generator.

the sub-model that is associated with the macro `Generate Documentation`. It is visible by the underlined label of `Generate Index Header` that this sub-model again has one entry point. For proper execution semantics, we also need to specify at which points the sub-model can be left in order to return to the parent model. This is done by means of model branches, which, in analogy to SIBs, define the "exits" of models. In jABC's graphical modeling tool, these exits are not visualized by concrete edges pointing to the parent model, but the information is displayed in one of the inspectors (the "Graph" inspector). For illustration, Fig. 3 indicates the model exits via dashed arrows. In the example, each SIB instance contained in the sub-model has a branch labelled "error", all of them exported and mapped to a model branch which is also called "error" (1)[3]. This means that the error handling for all execution steps in the sub-model is delegated to the parent model. Furthermore, the SIB instance `Write Index Page` exports its "default" branch as a model branch, which is also called "default" (2). In the parent model, the macro `Generate Documentation` provides exactly those two exits "default" and "error", which are defined as model branches in the underlying sub-model. As with normal SIBs, these branches then can be assigned to outgoing edges of the macro: For the application expert, there is no difference in using a SIB or a macro. Likewise, it is possible to define model parameters of a sub-model, which then become the parameters of an associated macro.

The fact that SLGs can be modeled hierarchically is key to both the clarity and the reusability of models. The appropriate use of sub-models improves the legibility and maintainability of SLGs and keeps them from getting too big and complex. Furthermore, especially thanks to model parameters, models can be reused just like SIBs, which forms the basis for the feature library described above (cp. Fig. 1).

2.3 jABC Plugins

The jABC framework provides a tool for graphically modeling SLGs. The functional range of the tool can be extended by plugins. For instance, the verification of constraints or the integration of Genesys' code generators are realized as jABC plugins.

The *Tracer* [2] is another very important jABC plugin. It is an interpreter that enables the user to execute (animate) a model *in the modeling tool*, to e.g. support rapid prototyping. In order to be executable by the Tracer, SIBs have to provide a service adapter implemented in Java.

Another example of a plugin is the *AnnotationEditor*, which allows to annotate almost any kind of information to jABC projects and to any SLG constituent, including SIBs, macros, features and the SLG itself. In the tutorial presented here, the AnnotationEditor is used for attaching HTML-formatted documentation to SLGs and the contained SIBs.

[3] The name of a model branch/model parameter can be defined freely by the application expert.

3 Building Code Generators with Genesys

Genesys [1] is a framework for constructing code generators based on jABC. More precisely, it is a special jABC bundle that provides tools (the "Genesys Developer Tools") and SIB libraries in order to support modeling code generators as SLGs. Building code generators as models offers several advantages [1], such as the acceleration of development due to a high degree of reusability, or the amenability to formal methods like model checking. Currently, Genesys' main focus is the construction of code generators for jABC itself, though we are currently researching the general applicability of this approach (e.g. for generating code from Ecore models, see Sect. 6). However, the tutorial in this paper will focus on a code generator for jABC, i.e. that it takes SLGs as its input, in order to demonstrate all perspectives and roles that are possibly using Genesys.

Especially for jABC code generators constructed with Genesys, the framework also provides a library of ready-made code generators for translating SLGs to different target languages and platforms like e.g. Java, C#, BPEL, Android, iPhone OS etc. Those generators are made available to jABC's application experts via a corresponding plugin (the "Genesys jABC Plugin").

Thus there are at least two different types of users for Genesys: the domain experts and the application experts. *Domain experts* (who often are developers and sometimes even congruent with the SIB experts), who have deep knowledge of the target domain for which a particular application should be modeled with jABC. Their main task is to customize jABC in a way that it fits the domain. This includes organizing SIBs and FLGs in a taxonomical structure, selecting appropriate plugins and defining how models are translated to code in that domain (by creating new code generators with Genesys or selecting existing ones). *Application experts*, which already have been introduced in Sect. 2.1, then use the jABC variant customized by the domain experts to model and generate an application for the target domain.

As the main contribution of this paper, we now show in detail how a code generator is built using the Genesys approach, in a tutorial-like fashion.

4 Tutorial: The Documentation Generator

In the following sections we will demonstrate the construction of a complete code generator with Genesys. This generator produces an HTML documentation website from a set of jABC models. We will show how to profit from reusing existing models and SIBs, and we will examine the construction process from the perspective of both the domain expert and the application expert. Note that as implementing new SIBs is in most cases not necessary for building code generators, we will not elaborate on the work of the SIB expert in this tutorial.

Requirements: The Documentation Generator. As we are constructing a code generator that is to be used in jABC, the Documentation Generator takes jABC models as its input. The generator's task is to produce an HTML

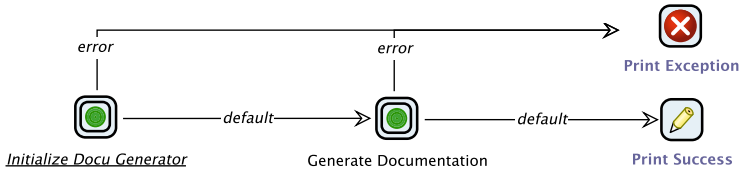


Fig. 4. The `Docu Generator` main model (topmost hierarchy level)

documentation website (comparable to the Javadoc Tool) from those models according to the following requirements:

1. The generator should process all jABC models in a given directory.
2. For each jABC model, a separate HTML page should be generated, containing the following information:
   - the documentation of the model and
   - a list of all SIB instances contained in that model. Each list entry should display the corresponding SIB's label. Furthermore, each entry should be linked to a detail page (described in 3) containing the documentation of the particular SIB instance, as well as to the corresponding public SIB documentation [10].
3. For each SIB instance in each jABC model, a detail HTML page will be generated, displaying the SIB instance's documentation. This page should be linked to the corresponding model page.
4. An index page should be generated, listing all processed models along with links to their respective model pages.
5. Each generated HTML page should contain a timestamp in order to retain the time of the last generation.

From the user perspective, the Documentation Generator should be easily usable via the Genesys jABC plugin.

4.1 Modeling the Code Generator

Based upon the requirements described above, we can now start modeling the code generator. As outlined in Sect. 3, this task is usually performed by a domain expert who is versed in the target domain as well as in using jABC and Genesys. For the Documentation Generator, the target domain comprises jABC models along with their associated concepts (SIBs, branches etc.), and the HTML format.

In the following, we will show how to model the Documentation Generator in jABC. For the sake of simplicity, we will not elaborate on all the parameterizations of the employed SIBs in detail. Instead we will focus on which SIBs are used to solve the task and how they are connected to each other. For each used SIB we will name the corresponding class so that it is easily possible for the reader to reproduce the example by himself. If no class is named, then the SIB class is equal to the SIB label displayed in the model. Detailed information for all SIB classes can be found in the online SIB documentation [10].

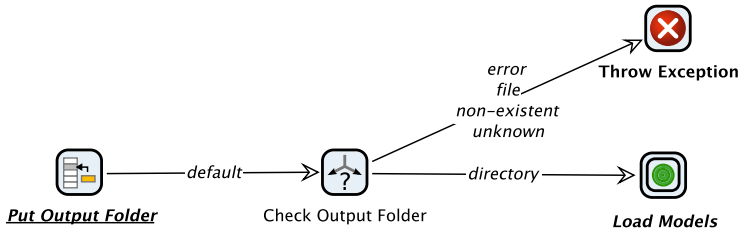


Fig. 5. The `Initialize Docu Generator` Model (second hierarchy level)

Structuring the Generation Process: Modeling a code generator with Genesys proceeds top-down. Thus as the first step of modeling the code generator, we divide the generation process into two abstract coarse-grained phases: the *initialization phase* and the *generation phase*. In the initialization phase, the code generator will set up the generation by verifying the input parameters and loading the jABC models, and in the generation phase the HTML website is produced.

Fig. 4 shows the resulting model, containing the SIB `Initialize Docu Generator` for the initialization phase and `Generate Documentation` for the generation phase. Both SIBs are macros (SIB class `MacroSIB`), as both phases will be refined and concretized in the following. Along with the two macros, the model contains two other SIBs emitting either a success message (`Print Success`, SIB class `PrintConsoleMessage`) when the two phases have been finished successfully, or an error message (`Print Exception`) if anything failed during the execution of the code generator.

Note that in our example, the error handling is always delegated to the main model depicted in Fig. 4. All the SIBs used in the Documentation Generator's models have "error" branches that lead to the SIB `Print Exception`, either as direct edges in the main model, or as model branches in all the other models. Consequently, `Print Exception` is the central (though very simple) error handling step for the entire generator.

The Initialization Phase: We proceed by concretizing the initialization phase. Basically, this phase has to verify the input parameters provided by the user and to set up the generation process. The Documentation Generator will have two input parameters:

outputFolder, the absolute path to the output directory for the generated HTML files, and

modelPath, a list of absolute paths to directories containing the jABC models for which the documentation should be generated.

The refined model for the initialization phase is depicted in Fig. 5.

We start by processing the generator parameter "outputFolder", whose value is first put into the execution context (`Put Output Folder`, SIB class `PutFile`) in order to be accessible by the following SIBs. Afterwards, the SIB `Check`

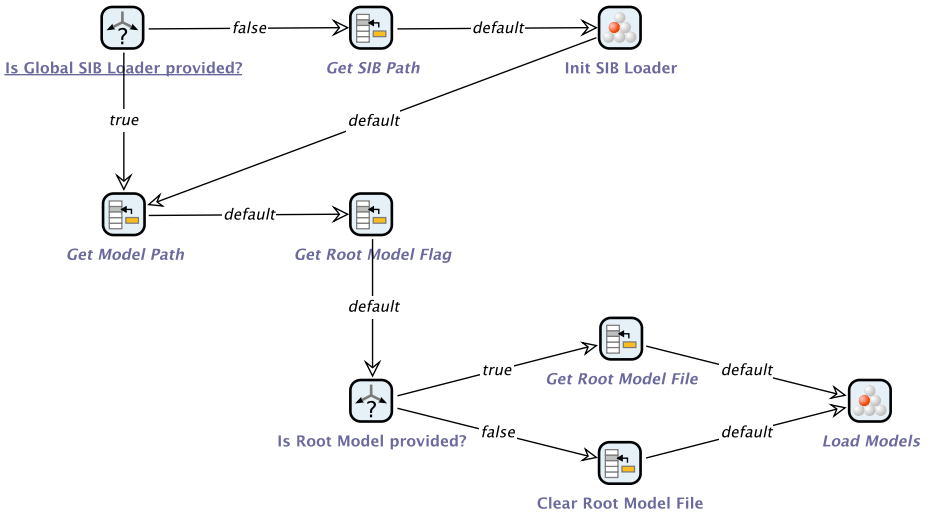


Fig. 6. The `Load Models` Model (third hierarchy level)

`Output Folder` (SIB class `CheckPath`) verifies this value: If it does not denote a proper (i.e. existent and writeable) directory, the following step `Throw Exception` issues an error. Otherwise, the initialization phase continues with handling the second input parameter "modelPath" (macro `Load Models`), which is again performed in a sub-model.

The sub-model referenced by `Load Models` is displayed in Fig. 6. As loading models is a standard task performed by all Genesys code generators, this model is part of the feature library and thus can be entirely reused for the Documentation Generator, without any changes. A detailed discussion of this model loading process is not required for this paper: The domain expert does not have to know the details of model loading in jABC anyway, as from his perspective, the ready-made model is used just like a SIB.

The Generation Phase: For modeling the generation phase, we go back to the main model (Fig. 4) and refine the macro `Generate Documentation`. The resulting model is depicted in Fig. 7.

The generation process starts with generating the static header of the index page (`Generate Index Header`) using StringTemplate. Note that all SIBs with "ST" on their icon are instances of the SIB class `RunStringTemplate` described in Sect. 2. The header of the index page only consists of static text, for instance containing the opening `html` and `body` tags for the document. Afterwards, a time stamp is generated (`Generate Time Stamp`, SIB class `GetTimeStamp`), which is inserted into the footer of each generated HTML page. The generation of the index page content and the detail pages for the models and the contained SIBs is again modeled in a sub-model (referenced by the macro `Generate Model Pages`). After the detail pages are produced, the generator finalizes the index

Fig. 7. The `Generate Documentation` Model (second hierarchy level)

page. For this purpose, the SIB `Generate Index Footer` is parameterized with the following simple template:

```
    </ul>
    <hr>
    <i>Generated: $timeStamp$</i>
  </body>
</html>
```

Besides some closing tags, this template contains a placeholder called "timeStamp", which is enclosed by dollar signs. When the generator is executed, StringTemplate replaces this placeholder by the timestamp produced by the step `Generate Time Stamp`. The generation phase finishes with writing the index page to a file (`Write Index Page`, SIB class `WriteTextFile`).

The sub-model that refines the macro `Generate Model Pages` is depicted in Fig. 8. It starts by iterating all jABC models that have been loaded in the initialization phase (`Next Model`, SIB class `IterateElements`).[4] As long as there are still models left to be processed, the "next" branch of the SIB will be used, otherwise the execution proceeds with the parent model (Fig. 7), connected via a model branch. The following step `Update Model Counter` (SIB class `UpdateCounter`) keeps track of a model number that is incremented each time the SIB is executed. This number is required to construct the names for the model detail pages. Then the generator extracts some information from the current model: The SIB `Get Model Name` stores its name in the execution context, and the SIB `Convert Content to Html` reads the documentation annotated via the AnnotationEditor and converts it to proper HTML markup, which is also stored in the execution context. Now the generator has gathered enough information for generating an index page entry for the current model (basically the model name, linked to the model detail page, step `Generate Index Entry`), as well as the header of the model detail page, containing the model's documentation and name.

To generate the list of SIBs in the current model along with the SIB detail pages, the generator retrieves all contained SIB instances (`Get SIB Graph Cells`) and then again delegates the production of all SIB-specific HTML markup to a sub-model (macro `Generate Markup for SIBs`). Finally, the footer of the detail page is generated (`Generate Model Page Footer`) and the entire page is written to a file (`Write Model Page`).

[4] Please note that the Documentation Generator produces HTML pages for *all* jABC models in a given directory, which particularly includes all used sub-models. Consequently, we do not need to expand the macros or to use any recursion - a simple iteration of the models is sufficient.

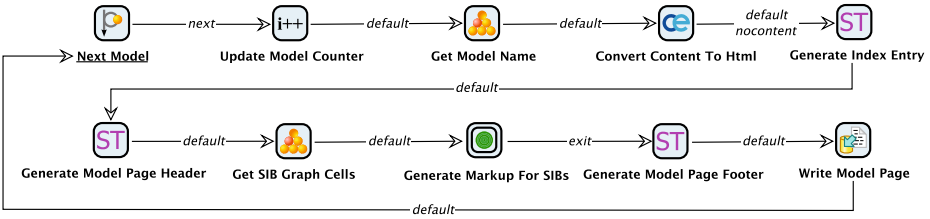


Fig. 8. The `Generate Model Pages` Model (third hierarchy level)

The last model required for the Documentation Generator refines the macro `Generate Markup for SIBs` (see Fig. 9). In the first step, it iterates all SIB instances contained in the current model (`Next SIB Graph Cell`, SIB class `IterateElements`), which works just like the `Next Model` step in Fig. 8. Furthermore, analogous to the model detail pages, the SIB `Update SIB Counter` (SIB class `UpdateCounter`) keeps track of a SIB counter that is used for the file names of the resulting SIB detail pages. Then again, some information is collected from the current SIB found in the execution context: its class name (`Get SIB Class Name`), unique identifier (`Get SIB Class Name`) and instance label (`Get SIB Label`).

The following SIB `Generate Documentation Link` differs from the other SIBs used in the Documentation Generator, as it is the only one that calls a remote functionality, in this case a web service available on the internet. This web service takes a SIB's class name and UID as input and uses this information to construct the URL of the corresponding online SIB documentation [10]. As this SIB was not provided by the ready-made SIB libraries, it had to be newly implemented. An appropriate service adapter only needed to realize the communication with an already existing web service, so that this implementation was very easy, which we will demonstrate in the following Sect. 4.2.

In the following step, the code generator reads the current SIB's documentation, attached via the AnnotationEditor (`Convert Content To Html`). Depending on whether such an annotated documentation could be found, a list entry for the current SIB on the model page is generated. In case a documentation is found, this entry is linked to a SIB detail page which is generated in the step `Generate SIB Page`. This SIB is parameterized with the following template:

```
<html>
  <body>
    $sibDoc$
    <a href="model_$modelCounter$.html">back to "$modelName$"</a>
    <hr>
    <i>Generated: $timeStamp$</i>
  </body>
</html>
```

Again, the static text contains placeholders that are replaced by StringTemplate, using information collected by the code generator:

sibDoc: The current SIB's HTML documentation retrieved by the `Convert Content To Html` step in Fig. 9.

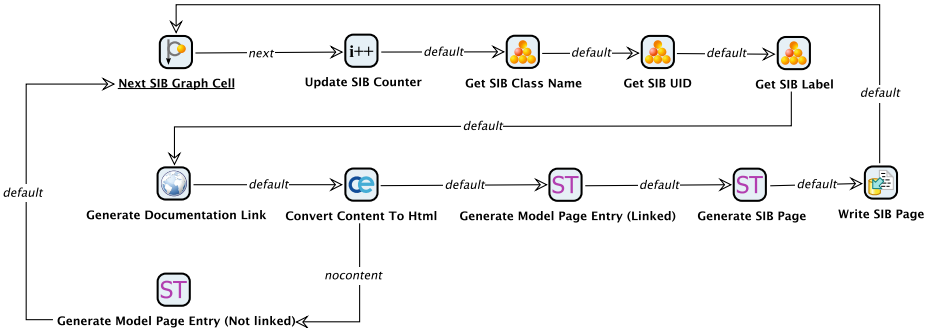


Fig. 9. The `Generate Markup for SIBs` Model (fourth hierarchy level)

modelCounter: The number of the current model assigned by the SIB `Update Model Counter` in Fig. 8.

modelName: The name of the current model retrieved by the SIB `Get Model Name` in Fig. 8.

timeStamp: The time stamp produced by `Generate Time Stamp` in Fig. 7.

As clearly visible from the usage of this information in the template, SIB instances in sub-models can easily access information left in the execution context by SIB instances at arbitrary levels of the model hierarchy. This is due to the global nature of the execution context: There is only one single context which is shared by all SIB instances across the entire model hierarchy.

Finally, if a SIB detail page has been generated, it is also written to a file (`Write SIB Page`).

Summary: We have now modeled a complete code generator according to the requirements listed above. The resulting generator consists of 6 models (5 new, 1 could be reused from the feature library), containing 43 SIB instances (of 23 different SIBs). Only one SIB had to be implemented, as the rest of the required functionality could be covered with existing SIBs. The depth of the resulting model hierarchy is 4.

4.2 Implementing the `Generate Documentation Link SIB`

In order to get an impression of how SIBs are implemented, this section briefly shows the anatomy of the SIB `Generate Documentation Link`. As already mentioned above (Sect. 4.1), this SIB uses a web service available on the internet in order to produce the URL of the corresponding online documentation from a given SIB's class name and unique identifier.

First, we specify the SIB itself, which is described via a simple Java class shown in Listing 1.1. The implementation starts with the `@SIBClass` annotation, which marks the class as a SIB and defines its unique identifier. Afterwards, the SIB's constituents are defined as fields:

- The SIB's branches "default" and "error" are specified via the `BRANCHES` array (line 4).
- The inputs (`sibClassNameKey`, `sibUidKey`) and output (`resultKey`) are defined as public fields. The given strings represent execution context keys for reading or storing the corresponding data (lines 7-13) that can be customized by the application expert.

The implementation of the interface `Generatable` marks the SIB as having a Java implementation that can be used for code generation. The interface demands the method `generate`, which defines the name of the corresponding service adapter (`WebServiceAdapter`), the name of the method which realizes the SIB's behavior (`generateDocumentationLink`) and the names of the SIB parameters that should be passed to the service adapter. In order to add service adapters for other target platforms, corresponding interfaces are added to the SIB. For instance, if the SIB required an implementation in Ruby, it would implement an additional interface `RubyGeneratable`, demanding a method `generateRuby` that provides all information necessary to generate calls to a Ruby script.

```
1  @SIBClass("webservices/GenerateDocumentationLinkSIB")
2  public class GenerateDocumentationLinkSIB implements Generatable {
3    // SIB branches
4    public static final String[] BRANCHES = { "default", "error" };
5
6    // execution context key for the SIB's class name
7    public String sibClassNameKey = "sibClassName";
8
9    // execution context key for the SIB's unique identifer
10   public String sibUidKey = "sibUid";
11
12   // execution context key for the result
13   public String resultKey = "result";
14
15   public ServiceAdapterDescriptor generate() {
16     return new ServiceAdapterDescriptor("WebServiceAdapter", "
         generateDocumentationLink", "sibClassNameKey", "sibUidKey",
         "resultKey");
17   }
18 }
```

Listing 1.1. Implementation of the SIB `Generate Documentation Link`

As the second step, we have to implement the service adapter that realizes the SIB's behavior. Listing 1.2 shows the corresponding service adapter for Java. Again, it is a simple Java class, that contains a static method providing the actual implementation. The name of the class, the name of the method and the order of the method's parameters have to correspond to the information given in the SIB's `generate` method. As an exception, the parameter `environment` (line 2), which is the current execution context (cp. Sect. 2.1), always has to be the first parameter as a convention, and thus does not have to be defined in the SIB. The actual implementation of the method is very simple:

- In lines 4 and 5, the SIB's class name and unique identifier are retrieved from the execution context, using the given context keys.

- Afterwards, an instance of the web service is created (line 7). The corresponding classes are generated from the web service's public WSDL via the JAX-WS framework [11].
- In line 8, the web service is called and its result is written into the execution context, using the given context key.
- Finally, depending on whether the web service invocation was successful or not, the method returns the "default" or the "error" branch.

```
1  public class WebServiceAdapter {
2    public static String generateDocumentationLink(
         LightweightExecutionEnvironment environment, String
         sibClassNameKey, String sibUidKey, String resultKey) {
3      try {
4        String sibClassName = (String) environment.getLocalContext().get
             (sibClassNameKey);
5        String sibUidName = (String) environment.getLocalContext().get(
             sibUidKey);
6
7        GenerateDocumentationLinkService port = new
             GenerateDocumentationLinkWebServiceService().
             getGenerateDocumentationLinkServicePort();
8        environment.getLocalContext().put(resultKey, port.
             generateDocumentationLink(sibClassName, sibUidName));
9      } catch (Exception exp) {
10       return "error";
11     }
12     return "default";
13   }
14 }
```

Listing 1.2. Java Service Adapter for the SIB `Generate Documentation Link`

4.3 Generating the Code Generator

While modeling a code generator, it is possible at any time to execute, debug and test it using jABC's Tracer. However, for productive use of the code generator, it should be translated to code itself, for instance in order to be able to use it via the Genesys jABC Plugin. This is usually performed in two steps: *editing the generator's meta data* and finally *generating the generator*.

In Genesys, the meta data describing a code generator is accumulated in a so-called *descriptor*. For instance, this contains a name for the code generator, a short and a long description (the latter should e.g. provide information about the generator's parameters), the name of the author, a version number, a classification of the generator as an Extruder or a Pure Generator (see [1]), an icon etc. For editing a generator's descriptor, the Genesys Developer Tools provide a special inspector, depicted in Fig. 10 (left).

Besides bundling meta data, the descriptor also has an important technical purpose for the next step, the generation of the code generator. When translating a model to code, Genesys' code generators use the existence of a descriptor as a trigger to determine whether it is a code generator that is being generated, or anything else. In the first case, some extra code is produced, e.g. the implementation of a special interface called `CodeGenerator`. Later on, this allows the Genesys jABC Plugin to detect the code generator and to make it available to the user.

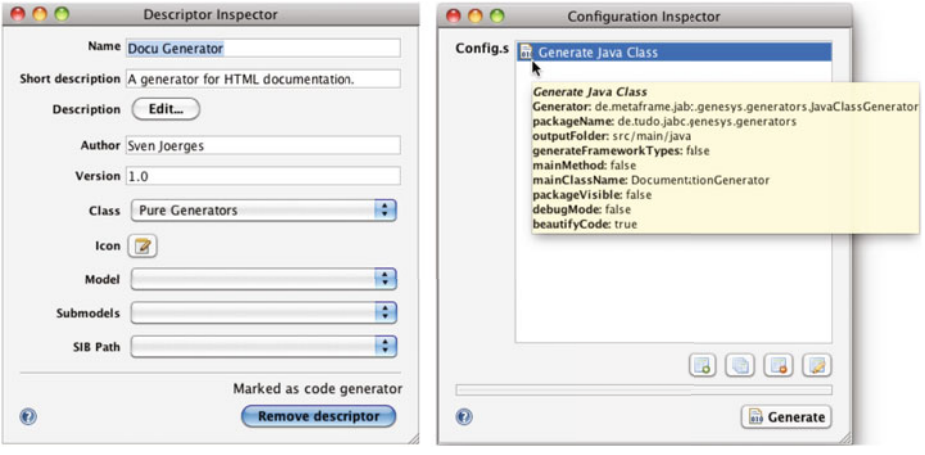


Fig. 10. Left: The inspector for editing a code generator's meta data. Right: Creating a code generator configuration.

After specifying the meta data and thus declaring the model to be a code generator, we can proceed with translating it to executable code. This is performed just like for any other model, i.e. using the Genesys jABC Plugin to create a generator configuration. Creating such a configuration always includes selecting an appropriate code generator for the translation, and then properly configuring it. In our example, we select the "Java Class Generator" (that generates simple executable Java classes [1]). Fig. 10 (right) shows the inspector for creating configurations, and it is visible in the tooltip of the selected configuration how the "Java Class Generator" has been parameterized to generate the Documentation Generator. After the configuration is created, the generator can be translated to Java source code.

To finally make the Documentation Generator available to the Genesys jABC Plugin (and thus to the application expert), some manual steps need to be performed by a person with technical skills, i.e. a developer or administrator. In particular, the generated source has to be compiled with a Java compiler and added to jABC's classpath (e.g. as a JAR file). Finally, the user's jABC instance has to be restarted in order to load the new code generator. We are currently working on automating this process, so that a new code generator can be generated, compiled and loaded into jABC on-the-fly, without interrupting the domain expert's workflow (cp. Sect. 6).

4.4 The Application Expert's Perspective: Using the Documentation Generator

After the Documentation Generator has been modeled, generated and loaded as described in Sect. 4.1, its usage is very simple. For an arbitrary jABC model, the application expert is now able to generate an HTML documentation by creating a generator configuration: Fig. 11 (left) shows the corresponding dialog. After

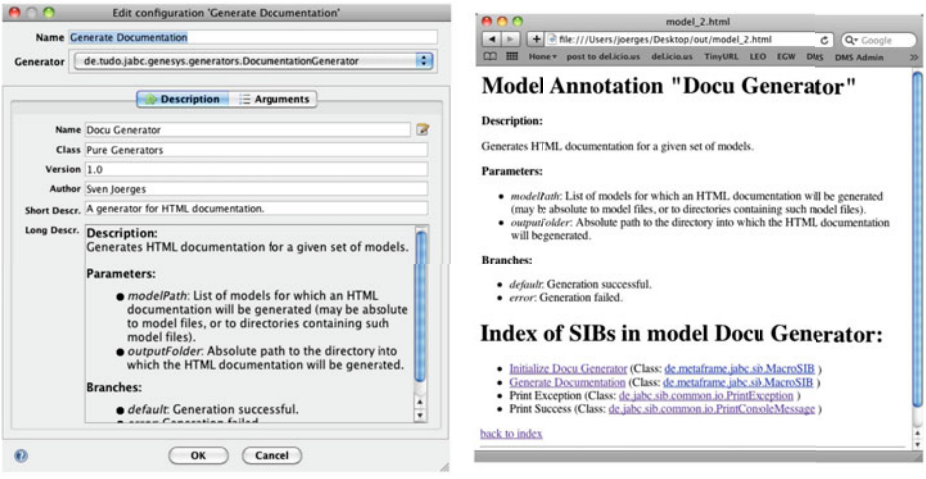


Fig. 11. Left: Creating a generator configuration with the Documentation Generator, Right: HTML documentation produced by the Documentation Generator

selecting the Documentation Generator, the meta data specified by the domain expert (see Sect. 4.3) is displayed and the generator can be configured. This new configuration can then be used to invoke the Documentation Generator. An exemplary part of a generated HTML documentation page is depicted in Fig. 11 (right): in this case, the Documentation Generator generated its own documentation.

5 Related Work

A lot of related solutions aim at supporting the development of code generators or model-to-text (M2T) transformations, including M2T languages like JET [12], the Epsilon Generation Language (EGL, [13]) or MOFScript [14], template engines like Velocity [15] or StringTemplate [9], or full-blown code generation frameworks like e.g. openArchitectureWare [16] or AndroMDA [17]. Most of these solutions are tightly coupled with their own template language, e.g. the Velocity Template Language (VTL, [15]) for Velocity or XPand [18] for oAW (AndroMDA is an exception, as it provides facades for changing the underlying template language).

Genesys does neither restrict to nor define a particular M2T or template language. Due to its service-oriented approach, almost any M2T tool or template engine can be used for Genesys code generators: The integration has to be performed once by a SIB expert who implements corresponding SIBs. Currently, Genesys already ships with ready-made SIBs for e.g. Velocity and StringTemplate. As a SIB's granularity is not restricted to libraries or small software components

(cp. Sect. 2.1), also entire code generation frameworks can be incorporated into Genesys this way. For instance, we integrated AndroMDA as SIBs, so that Genesys is able to generate code for UML diagrams. Due to this flexibility, we also consider Genesys a *meta-framework*.

Another important concern of Genesys is the clear separation of the *generation logic* (traversing a model's elements, collecting data etc.) from the actual *output definition* (e.g. resulting code or markup). Most of the M2T/template languages support the usage of control flow statements like loops or conditions, and sometimes even calling external code via special directives. Although being powerful mechanisms, such features lead to mixing generation logic and output definition, in our opinion, at the expense of readability and maintainability. In Genesys, anything related to the generation logic should be reflected in the model, while the output definition, i.e. the templates, should not contain more than simple placeholders (variables). For calling external code, corresponding SIBs should be implemented (if they do not already exist) and integrated into the code generator model. Although this approach demands a certain discipline from the code generator developer, there are several benefits from having the entire generation logic as a formal model, such as verifiability.

Some of the tools and frameworks named above support the invocation of generators and templates as parts of a workflow. For instance, oAW provides an XML-based workflow language, and EGL can be used in Apache Ant [19] scripts. These workflows are in most cases used to build tool chains, but never to separate the generation logic from the output definition. Furthermore, the underlying workflow definitions are not amenable to verification with formal methods. In Genesys, formal models are used both for the generation logic and, at a higher hierarchy level, for building tool chains.

Concerning verifiability, there are various approaches that also aim at assuring the reliability of automatic code generation. A "verifying compiler", as proposed by Tony Hoare and Jay Misra in their grand challenge [20], is one possible solution to achieve this goal. According to this approach, the compiler is able to determine the correctness of the problem by using additional information, like e.g. assertions or annotations, in the source code. A lot of research has been done on this, e.g. taking the form of proof-carrying code [21] or evidence-based approaches [22]. Other work aims at verifying the compiler itself, like e.g. the Verifix project [23], or at validating the translation especially for optimizing compilers [24]. In contrast to these rather *analytical* techniques, *constructive* approaches postulate a more systematic development process, e.g. based on the adoption of accepted standards [25] or the generation of code from specifications [26]. All these approaches have in common that they work more or less on the *source code level*. As all code generators in the Genesys framework are modeled as SLGs, the verification is applied on the *modeling level*, which is one of the key ideas of the XMDD paradigm. This also allows us to formulate constraints which are not only applicable to one code generator, but to whole families of code generators.

6 Conclusion and Future Work

In this paper, we have demonstrated how to build a complete code generator from scratch using the Genesys framework. We constructed the Documentation Generator, which is able to produce an HTML documentation from jABC models. While modeling, we profited from existing models and SIBs, which were reused for the code generator, supporting the observations we described in [1]. We also have demonstrated the flexibility of the SIB concept: Whether the functionality realized by a SIB is simple or complex, local or remote, implemented in Java or C, is entirely transparent to the domain expert, thus helping him to exclusively focus on the code generation logic. Furthermore, we examined the use of Genesys from the perspective of both the domain expert and the application expert.

Currently, we focus on applying Genesys to other inputs than SLGs to broaden the scope beyond jABC. Again, we profit from the framework's service-orientation: For traversing and collecting information from SLGs, we use a special SIB library, which can be easily replaced by a library of SIBs supporting other model types. All other SIBs that are used for Genesys (e.g. the SIBs for integrating template engines) are completely independent of the given model. For instance, we currently investigate the automatic generation of corresponding SIBs for a given Ecore [27] meta-model, which can then immediately be used to build a code generator for models that belong to this meta-model.

Furthermore, we are working on further simplifying the domain expert's work of building a code generator. As pointed out in Sect. 4.3, there are still some manual steps required to make a modeled code generator accessible for the application expert. Currently, the generated source has to be compiled with a Java compiler, added to jABC's classpath and afterwards, jABC has to be restarted in order to load the new code generator. In an upcoming version of the Genesys Developer Tools, this process will be entirely automated, so that the domain expert's workflow is not interrupted anymore.

References

1. Jörges, S., Margaria, T., Steffen, B.: Genesys: Service-Oriented Construction of Property Conform Code Generators. Innovations in Systems and Software Engineering 4(4), 361–384 (2008)
2. Steffen, B., Margaria, T., Nagel, R., Jörges, S., Kubczak, C.: Model-Driven Development with the jABC. In: Bin, E., Ziv, A., Ur, S. (eds.) HVC 2006. LNCS, vol. 4383, pp. 92–108. Springer, Heidelberg (2007)
3. Margaria, T., Steffen, B.: Service Engineering: Linking Business and IT. IEEE Computer 39(10), 45–55 (2006)
4. Technische Universität Dortmund: jABC Website (2009), `http://www.jabc.de`
5. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2001)
6. Margaria, T., Steffen, B.: Agile IT: Thinking in User-Centric Models. In: Proc. ISoLA 2008. CCIS, vol. 17, pp. 493–505. Springer, Heidelberg (2008)

7. Object Management Group: Model Driven Architecture, http://www.omg.org/mda/
8. Technische Universität Dortmund: Common SIBs Website (2010), http://www.jabc.de/sib
9. Terence Parr: StringTemplate Website (2009), http://www.stringtemplate.org/
10. Technische Universität Dortmund: jABC SIBs Website (2009), http://www.jabc.de/sib
11. Oracle: JAX-WS Reference Implementation (2010), https://jax-ws.dev.java.net/
12. Eclipse Foundation: JET, part of Eclipses Model To Text (M2T) component (2010), http://www.eclipse.org/modeling/m2t/?project=jet#jet
13. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.: The Epsilon Generation Language. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 1–16. Springer, Heidelberg (2008)
14. Oldevik, J., Neple, T., Grønmo, R., Aagedal, J.Ø., Berre, A.J.: Toward Standardised Model to Text Transformations. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 239–253. Springer, Heidelberg (2005)
15. Apache Software Foundation: Velocity Website (2007), http://velocity.apache.org/
16. openArchitecureWare Team: openArchitectureWare Website (2009), http://www.openarchitectureware.org/
17. AndroMDA Team: AndroMDA Website (2009), http://www.andromda.org/
18. Eclipse Foundation: XPand, part of Eclipses Model To Text (M2T) component (2010), http://www.eclipse.org/modeling/m2t/?project=xpand#xpand
19. Apache Software Foundation: Apache Ant Project (2010), http://ant.apache.org/
20. Hoare, C.A.R.: The Verifying Compiler: A Grand Challenge for Computing Research. J. ACM 50(1), 63–69 (2003)
21. Necula, G.C.: Proof-Carrying Code. In: Proc. POPL 1997, pp. 106–119. ACM Press, New York (1997)
22. Denney, E., Fischer, B.: Extending Source Code Generators for Evidence-based Software Certification. In: Proc. ISOLA 2006, pp. 138–145 (2006)
23. Goos, G., Zimmermann, W.: Verification of Compilers. In: Olderog, E.-R., Steffen, B. (eds.) Correct System Design. LNCS, vol. 1710, pp. 201–230. Springer, Heidelberg (1999)
24. Necula, G.C.: Translation Validation for an Optimizing Compiler. ACM SIGPLAN Notices 35(5), 83–94 (2000)
25. Stürmer, I., Weinberg, D., Conrad, M.: Overview of existing safeguarding techniques for automatically generated code. In: Proc. SEAS 2005, pp. 1–6. ACM Press, New York (2005)
26. Coglio, A., Green, C.: A Constructive Approach to Correctness, Exemplified by a Generator for Certified Java Card Applets. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 57–63. Springer, Heidelberg (2008)
27. Eclipse Foundation: Eclipse Modeling Project (2009), http://www.eclipse.org/modeling/

The Need for Early Aspects

Ana Moreira and João Araújo

Departamento de Informática, CITI/FCT, Universidade Nova de Lisboa,
2829-516 Caparica, Portugal
{Amm,ja}@di.fct.unl.pt

Abstract. Early aspects are crosscutting concerns that are identified in the early phases of the software development life cycle. These concerns do not align well with the decomposition criteria of traditional software development paradigms and, therefore, they are difficult to modularise. The result is their specification and implementation scattered along several base modules, producing tangled representations that are difficult to maintain, reuse and evolve. It is now understood that the influence of requirements that cut across other requirements results in incomplete understanding of specified requirements and limits the architectural choices. Thus, a rigorous analysis of crosscutting requirements and their interactions is essential to derive a balanced architecture. Early Aspects offer additional abstraction and composition mechanisms for systematically handling crosscutting requirements. This paper focuses on two pioneering requirements approaches, one based on viewpoints and another based on use-cases.

Keywords: Requirements analysis, separation of concerns, aspect-orientation.

1 Introduction

The established principles of Software Engineering [23], such as modularization, abstraction and encapsulation, play a major role in achieving separation of concerns [8]. Traditional Software Engineering methods, such as structured and object-oriented approaches, have been developed with those principles in mind. Nonetheless, the modularization techniques they provide cannot separate all interrelated complex concerns. Certain broadly-scoped properties (e.g., response time, security, persistence) are very difficult to modularize. This is because existing approaches generally follow a dominant decomposition criterion that is not suitable to capture and represent all kinds of concerns found in software applications [12]. This problem is known as the *tyranny of the dominant decomposition*: the system is modularized in only one way at a time and, consequently, the concerns that do not align with that decomposition criteria end up scattered across many modules [25]. Such *crosscutting concerns* span traditional module boundaries (for example, classes in an object-oriented decomposition), hindering understandability, maintainability and evolution.

Modern approaches propose mechanisms for decomposition and composition. However, they mostly use a dominant base decomposition, with other possible dimensions cutting across them. For example, approaches, such as the Non-Functional Requirements framework [5], use non-functional requirements as the dominant dimension with the functional dimension added a posteriori. Other requirements

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 386–407, 2011.

engineering (RE) approaches, such as viewpoints [9] and use cases [10], use functional requirements as the dominant decomposition with analysis conducted against a set of non-functional requirements cutting across the base. Crosscutting is a phenomenon that is not limited to Non-Functional Requirements (NFRs) and functional requirements can also often cut across parts of a system [20].

It is now understood that the influence of requirements that cut across other requirements result in incomplete understanding of specified requirements and limits the architectural choices. Thus, a rigorous analysis of crosscutting requirements and their interactions is essential to derive a balanced architecture. Early Aspects work with existing requirements approaches by offering additional abstraction and composition mechanisms for systematically handling crosscutting requirements.

This paper focuses on the identification and modelling of requirements-level crosscutting concerns in viewpoint and use-case based requirements models. An early identification of crosscutting concerns provides a means for: reasoning about the problem domain; handling quality attributes; performing trade-off analysis whenever conflicting situations are detected; supporting better architectural choices. Thus, we address Requirements Engineering and do not comment on which approach or technology should be used to implement the resulting specification.

This work aims at: clarifying the advantages of aspect-oriented requirements identification, modelling and analysis over existing techniques; discussing, by means of real-world examples, how one can identify, model, compose and analyse aspects at the requirements-level using relevant techniques and tools; discussing the role of aspectual requirements in the software lifecycle with regards to improving software modularity and associated quality attributes. A fresh contribution is the use of MATA to model volatile concerns, making composition much simpler and expressive compared to the previous approach.

This paper starts with an overview on the basic concepts of Early Aspects. Section 2 introduces the aspect-oriented concepts, and in particular the aspect-oriented requirements analysis methods. Then we choose two representative approaches, one being an extension of viewpoints (Section 3), and the other a use case based approach we developed having in mind requirements evolution Section 4) and finish comparing them. Finally, we present our conclusions and discuss future work.

2 Background

Aspect-Oriented Software Development (AOSD) appeared as a new step to improve the separation of concerns principle, which aims at identifying and modularizing those parts of software that are relevant to a particular concept, goal or purpose [8]. This is an essential principle in software engineering to reduce software development complexity and increase understandability, minimizing the impact of change through encapsulation of different concerns in separate modules. In the context of this work, a concern refers to a property that addresses a certain problem of interest to one or more stakeholders and which can be defined by a set of coherent requirements.

2.1 An Introduction to Aspect-Orientation

Software Engineering is continuously evolving, searching for more efficient techniques to modularise and compose software systems. Historically, most new development

techniques are introduced at the programming level and their concepts are later abstracted and applied at the earlier stages (e.g., design, analysis and requirements). We have seen this evolution tendency in the 70's, with structured techniques, and in the late 80's and early 90's, with object-oriented techniques [22].

AOSD is another step towards achieving improved modularity, aiming at modularizing crosscutting concerns by providing means for their systematic identification, separation, representation and composition [21]. Typical examples of crosscutting concerns are NFRs, but crosscutting concerns can also be functional requirements, such as auditing, or validation [14, 20]. Crosscutting concerns are encapsulated in separate modules, known as *aspects*, and composition mechanisms are later used to weave them back with other base (or core) modules, at loading time, compilation time, or run-time [3]. Similarly to what happened to previous software development approaches, AOSD was introduced first at the programming level. The concepts have then moved beyond programming and are now being applied at earlier development stages, such as design, architecture and requirements engineering [6].

2.2 Aspect-Oriented Requirements and Its Influence on Architecture Design

Aspect-oriented requirements techniques build upon the strong focus on *composition* in AOSD by providing a fine-grained specification of how a requirements-level aspect constrains or influences specific requirements in a system. Such a detailed understanding of the composition relationships between aspectual and non-aspectual requirements leads to an improved understanding of their interaction, inter-relationships and conflicts. This, in turn, helps to identify trade-offs early on in the development life cycle and undertake negotiations with the affected stakeholders. Furthermore, the aspectual requirements and their associated trade-offs can be traced to implementation to ensure that they have been preserved in line with the requirements specification that the stakeholders signed off on.

An aspect at the requirements-level is a broadly-scoped property, represented by a single requirement or a coherent set of requirements, that affects multiple other requirements in the system so that: it may constrain the specified behaviour of the affected requirements; it may influence the affected requirements in order to alter their specified behaviour. Figure 1 shows requirements-level aspects affecting multiple requirements.

During composition, conflicting situations may be found and resolved even before the architecture design is derived. A conflict is detected any time a contribution relationship between two concerns is negative. These contributions are unidirectional and can be positive, negative or "none". For example, *Response Time* contributes negatively to *Security* and positively to *Availability*. Whenever there is a negative contribution between candidate aspects we are faced with a conflicting situation if these apply to the same or overlapping sets of requirements (e.g., in a viewpoint or in a use case). These conflicting situations have an impact on the architecture design. The required trade-offs pull an architecture in various directions, leading to a number of architectural choices that would serve stakeholders' needs with varying levels of satisfaction. Figure 2 illustrates this idea.

We created a decision support system to handle other types of conflicts (e.g., involving disagreements between stakeholders) [4, 17, 18].

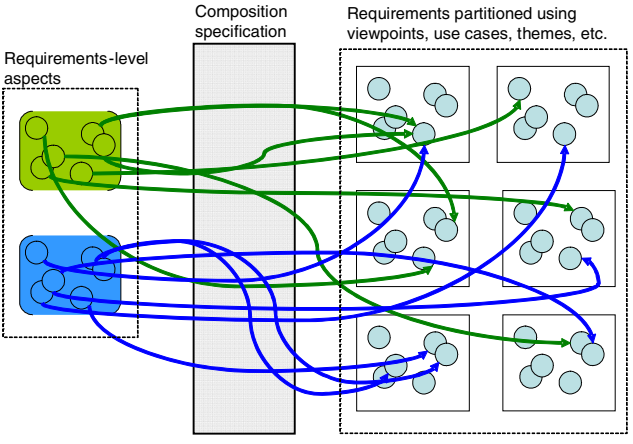


Fig. 1. Requirements-level aspects influencing other requirements [19]

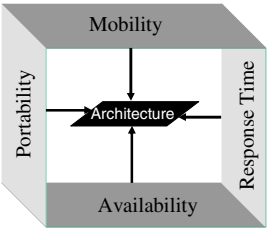


Fig. 2. Aspectual requirements pulling architecture in various directions [15]

2.3 Two Aspect-Oriented Requirements Approaches

We have chosen to discuss in the tutorial two pioneering and very different approaches, Arcade [21] and MATA [27], representing two different requirements paradigms: viewpoints and use-cases, respectively. Though viewpoints and use cases do provide subjective perspectives on a system, they do not treat non-functional properties (e.g., security, real-time, mobility) systematically. These properties often form good candidate aspects that cut across viewpoints and use cases. Some approaches treat NFRs explicitly, but do not handle their crosscutting nature nor do they handle their functional aspects. Finally, we selected two examples as we wanted to confront attendees with different modelling situations.

3 The Arcade Approach

Arcade supports separation of aspectual requirements, non-aspectual requirements and composition rules. This makes it possible to establish early trade-offs between aspectual requirements; hence, providing support for negotiation and subsequent decision-making among stakeholders.

3.1 Generic Model for AORE

Figure 3 illustrates the Arcade model [21].

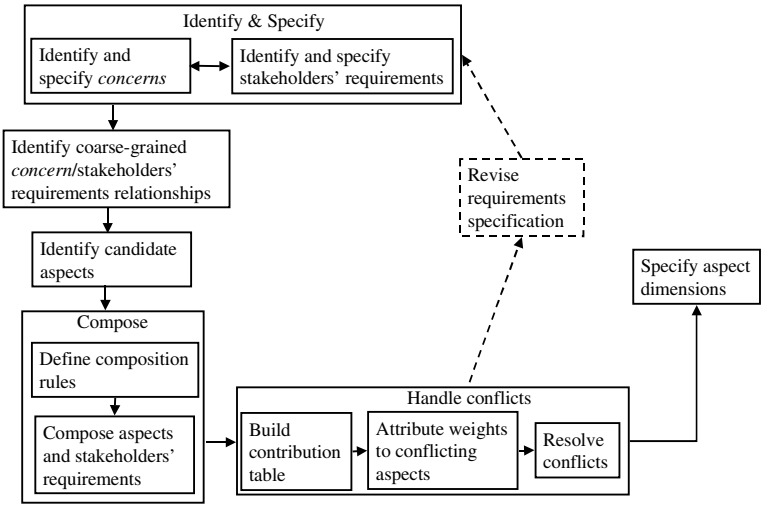


Fig. 3. Arcade model [21]

We start by identifying and specifying both concerns[1] and stakeholders' requirements. The latter is carried out using an existing requirements level separation of concerns mechanism such as viewpoints [9], use cases [10], goals [13] or problem frames [11]. The order in which the specification of concerns and stakeholders' requirements is accomplished depends on the dynamics of the interaction between requirements engineers and stakeholders. In any case, it is useful to relate concerns to requirements, through a matrix, as the former may constrain the latter. Once the coarse-grained relationships between concerns and stakeholders' requirements have been established and the candidate aspects identified, the next step is to define detailed composition rules. These rules operate at the granularity of individual requirements and not just the modules encapsulating them. Consequently, it is possible to specify how an aspectual requirement influences or constrains the behaviour of a set of non-aspectual requirements in various modules. At the same time, if desired, aspectual trade-offs can be observed at a finer granularity. This alleviates the need for unnecessary negotiations among stakeholders for cases where there might be an apparent trade-off between two (or more) aspects but in fact different, isolated requirements are being influenced by them. It also facilitates identification of individual, conflicting aspectual requirements with respect to which negotiations must be carried out and trade-offs established.

After composing the candidate aspects and stakeholders' requirements using the composition rules, identification and resolution of conflicts among the candidate

[1] The notion of *concern* here is that used in PREView, which has a direct correspondence with non-functional requirements. It is NOT the general concept as proposed by Dijkstra and used later in MATA.

aspects is carried out. Conflict resolution must lead to a revision of the requirements specification. If this happens, then the requirements are recomposed and any further conflicts arising are resolved. The cycle is repeated until all conflicts have been resolved through effective negotiations.

The last activity identifies the aspects' dimensions. We have observed that aspects at this early stage have an impact that can be described in terms of two dimensions:

- *Mapping*: an aspect might map onto a system feature/function (e.g. a simple method, object or component), decision (e.g. a decision for architecture choice) and design (and hence implementation) aspect (e.g. response time). This is why we call aspects at the RE stage candidate aspects as, despite their crosscutting nature at this stage, they might not directly map onto an aspect at later stages.
- *Influence*: an aspect might influence different points in a development cycle, e.g. availability influences the system architecture while response time influences both architecture and detailed design.

The concrete techniques we have chosen are viewpoints [9] for identifying the stakeholder requirements, and XML as the definition language for specifying these requirements, the candidate aspects identified and the composition rules to relate viewpoints with aspects. Tool support is provided by the Aspectual Requirements Composition and Decision support tool, Arcade (lending its name to the method). The tool makes it possible to define the viewpoint requirements, aspectual requirements and composition rules using pre-defined templates.

3.2 Case Study

The case study we have chosen is a simplified version of the toll collection system on the Portuguese highways [7]. "*In a road traffic pricing system, drivers of authorised vehicles are charged at toll gates automatically. The gates are placed at special lanes called green lanes. A driver has to install a device (a gizmo) in his/her vehicle. The registration of authorised vehicles includes the owner's personal data, bank account number and vehicle details. The gizmo is sent to the client to be activated using an ATM that informs the system upon gizmo activation.*

A gizmo is read by the toll gate sensors. The information read is stored by the system and used to debit the respective account. When an authorised vehicle passes through a green lane, a green light is turned on, and the amount being debited is displayed. If an unauthorised vehicle passes through it, a yellow light is turned on and a camera takes a photo of the plate (used to fine the owner of the vehicle). There are three types of toll gates: single toll, where the same type of vehicles pay a fixed amount, entry toll to enter a motorway and exit toll to leave it. The amount paid on motorways depends on the type of the vehicle and the distance travelled."

3.3 Illustrating Arcade with the Traffic Pricing System

3.3.1 Identify and Specify Stakeholders' Requirements

Based on the requirements described above, as well as in our knowledge as users of the system, the following viewpoints, some with sub-viewpoints, were identified [21]:

- *ATM*: allows customers to enter their own transactions using cards. The ATM sends the transaction information for validation and processing.

- *Vehicle*: enters and leaves toll gates. There is a sub-viewpoint Unauthorised Vehicle whose plate number is photographed.
- *Gizmo*: is read by the system and is glued on the windscreen of the car.
- *Police*: receives data about the unauthorised vehicles and their infractions.
- *Debiting System*: interacts with the bank to allow debits in clients' accounts.
- *Toll Gate*: through which the vehicles pass when entering or leaving the toll collection system. There are two sub-viewpoints: *Entry Toll* and *Paying Toll*. Entry Toll detects gizmos. The Paying Toll viewpoint is further refined into two sub-viewpoints: *Single Toll* turns the light green and displays the amount to be paid for authorised vehicles and turns the light yellow, sounds an alarm and photographs the plate numbers for unauthorised vehicles; *Exit Toll* behaves similarly to single toll, except that it must take into account the valid (or invalid) entrance of the vehicle.
- *Vehicle Owner*: who has three sub-viewpoints: *Registration* of vehicles, cancellation of registration and modification of registration details; *Billing* in the form of regular invoices; *Activation* of the gizmo using ATMs.
- *System administrator*: introduces new information and modifies existing information in the system.

Figure 4 shows a viewpoint in XML. The structure is self-explanatory. A Viewpoint tag denotes the start of a viewpoint while a Requirement tag denotes the start of a requirement. Refinements such as sub-viewpoints and sub-requirements are represented via the nesting of the tags. Each requirement has an id which is unique within its defining scope (the viewpoint). Viewpoint names are unique within each case study. However, XML namespaces can be used for the purpose as well.

```
<?xml version="1.0" ?>
- <Viewpoint name="Vehicle">
        <Requirement id="1">The vehicle enters the system when it is within ten meters of the toll gate.</Requirement>
        <Requirement id="2">The vehicle enters the toll gate.</Requirement>
        <Requirement id="3">The vehicle leaves the toll gate.</Requirement>
        <Requirement id="4">The vehicle leaves the system when it is twenty meters away from the toll
        gate.</Requirement>
        - <Viewpoint name="UnauthorisedVehicle">
            <Requirement id="1">The vehicle number plate will be photographed.</Requirement>
        </Viewpoint>
  </Viewpoint>
```

Fig. 4. The *Vehicle* viewpoint in XML

Note that currently this type of specification can be obtained by using text-mining supported by EA-Miner [24] in collaboration with RDL [26].

3.3.2 Identify and Specify Concerns

Concerns are identified by analysing the initial requirements. For example, since the owner of a vehicle has to indicate his/her bank details during registration, Security is an issue that the system needs to address. Other concerns in our case study, identified in a similar way, are: Response Time, Multi-Access System, Compatibility, Legal Issues, Correctness and Availability. For simplification we choose to provide the specification of only Response Time (Figure 5). The Requirement tag has the same

semantics and scoping rules as for the viewpoints. The only difference is that the defining scope is now the Concern. Like viewpoints, concerns can be nested as well and concern names are unique with the scope of a case study in Arcade.

```
<?xml version="1.0" ?>
- <Concern name="ResponseTime">
    - <Requirement id="1">
          The system needs to react in-time in order to:
              <Requirement id="1.1">read the gizmo identifier;</Requirement>
              <Requirement id="1.2">turn on the light (to green or yellow);</Requirement>
              <Requirement id="1.3">display the amount to be paid;</Requirement>
              <Requirement id="1.4">photograph the plate number from the rear;</Requirement>
              <Requirement id="1.5">sound the alarm;</Requirement>
              <Requirement id="1.6">respond to gizmo activation and reactivation.</Requirement>
      </Requirement>
  </Concern>
```

Fig. 5. The *Response Time concern* in XML

3.3.3 Identify Coarse-Grained Concern/Viewpoint Relationships

Now we can relate viewpoints and concerns, by building the matrix in Table 1.

3.3.4 Identify Candidate Aspects

Table 1 shows which concerns cut across specific viewpoints. For example, we can observe that the requirements in Response Time influence and constrain the requirements in the viewpoints: Gizmo, ATM, Toll Gate and Vehicle. Consequently, all concerns identified form candidate aspects as they cut across multiple viewpoints. In another system, a concern might constrain a single viewpoint and, hence, will not qualify as a candidate aspect (note that it will still be modularised as a concern).

Once a candidate aspect has been identified, the XML specification of the corresponding concern is transformed to reflect this fact. The transformation is a simple operation (using a simple transformation in XSLT – eXtensible Style Sheet Language for Transformations) which replaces the Concern tag with an Aspect tag. While this might seem a trivial transformation, it ensures that the specification reflects the aspectual nature of a concern.

3.3.5 Compose Aspects and Viewpoints: Define Composition Rules

Composition rules define the relationships between aspectual requirements and viewpoint requirements at a fine granularity. Composition rule definitions can be governed by an XML schema in Arcade. However, for simplification we describe the structure of composition rules with reference to some examples and not the XML schema definition. As shown in Figure 6, a coherent composition rule is encapsulated in a Composition tag, in this case, for Response Time requirements (partially). The semantics of the Requirement tag here differ from the tags in the viewpoint and aspect definitions. Each Requirement tag has at least two attributes: the aspect or viewpoint it is defined in and an id which uniquely identifies it within its defining scope. If a viewpoint requirement has any sub-requirements these must be explicitly excluded or included in the Constraint imposed by an aspectual requirement. This is done by providing an *include* or *exclude* value to the optional children attribute. A value of *all* for a viewpoint or id value implies that all the viewpoints or requirements within the specified viewpoint are to be constrained.

Table 1. Matrix relating *concerns* with viewpoints

(**P:** Police; **Gz:** Gizmo; **DS:** Debiting System; **TG:** Toll Gate; **PT:** Paying Toll; **ST:** Single Toll; **ExT:** Exit Toll; **ET:** Entry Toll; **Vh:** Vehicle; **UV:** Unauthorized Vehicle; **VO:** Vehicle Owner; **Act:** Activation; **Reg:** Registration; **Bill:** Billing; **Adm:** Administration)

VP / **Concerns**	*P*	*Gz*	*DS*	*ATM*	*TG*	*PT*	*ST*	*ExT*	*ET*	*Vh*	*UV*	*VO*	*Reg.*	*Act.*	*Bill.*	*Adm.*
Response Time		✓		✓	✓	✓	✓	✓	✓	✓	✓					
Availability		✓		✓	✓	✓	✓	✓	✓				✓	✓		✓
Security	✓		✓	✓								✓	✓	✓	✓	✓
Legal Issues	✓							✓					✓		✓	
Compatibility	✓		✓	✓										✓		
Correct-ness	✓	✓	✓		✓	✓	✓	✓	✓			✓	✓	✓	✓	
Multi Access		✓		✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓

The Constraint tag defines an often concern-specific action and operator defining how the viewpoint requirements are to be constrained by the specific aspectual requirement. Although the actions and operators are informal they must have clearly defined meaning and semantics to ensure valid composition of aspects and viewpoints. The Outcome tag defines the result of constraining the viewpoint requirements with an aspectual requirement. The action value describes whether another viewpoint requirement or a set of viewpoint requirements must be satisfied or merely the constraint specified has to be fulfilled.

```
<?xml version="1.0" ?>
- <Composition>
    - <Requirement aspect="ResponseTime" id="1.1">
        - <Constraint action="enforce" operator="between">
            <Requirement viewpoint="Vehicle" id="1" />
            <Requirement viewpoint="Vehicle" id="2" />
          </Constraint>
        - <Outcome action="satisfied">
            <Requirement viewpoint="Gizmo" id="1" children="include" />
          </Outcome>
      </Requirement>
  </Composition>
```

Fig. 6. The composition rules for *Response Time* requirements

The informality of the actions and operators ensures that the composition specification is still readable by the stakeholders, an important consideration during requirements engineering. For example, if we look at the composition rule in Figure 6 and focus on the values in bold we get the following: "Response Time requirement 1.1 must be enforced between requirements Vehicle 1 and Vehicle 2 with the outcome that Gizmo requirement 1 including its children is satisfied". The complete example and the specification and composition language can be found in [21].

3.3.6 Handling Conflicts

Aspects and viewpoints are composed using composition rules. This leads to identify conflicts among aspects whose requirements constrain the same or overlapping sets of viewpoint requirements. In Arcade, this process is optimised as any potential interaction or conflict can be deduced from the composition rules. Thus, one does not

need to compose the aspects and viewpoints until the conflicts have been resolved. XML semantic composition is now possible with [26].

Build the contribution table. This table shows in which way (negatively or positively) an aspect contributes to others. This matrix shown in Table 2 is symmetric, i.e., only the diagonally upper (or lower) triangle needs to be considered.

Table 2. Contribution matrix

Aspects / *Aspects*	*Response Time*	*Availability*	*Security*	*Legal Issues*	*Compatibility*	*Correctness*	*Multi-Access*
Response Time		+	–			–	–
Availability							+
Security						+	
Legal Issues					+	+	
Compatibility							
Correctness							
Multi-Access							

In this case, Response Time contributes negatively to Security, Correctness and Multiple Access and positively to Availability, for example. Whenever there is a negative contribution between aspects we are faced with conflict if these aspects apply to the same or overlapping sets of requirements in the viewpoints.

Attribute weights to conflicting aspects. To help resolve conflicts we allocate weights to the cells of the aspect/viewpoint matrix where the conflicting aspects apply to the same viewpoints. Weighting allows us to describe the extent to which an aspect may constrain a viewpoint (c.f. Table 3). The values are given according to the importance each aspect has for each viewpoint. The scales we are using are based on ideas from fuzzy logic and have the following meaning:

- Very important takes values in the interval] 0,8 .. 1,0]
- Important takes values in the interval] 0,5 .. 0,8]
- Average takes values in the interval] 0,3 .. 0,5]
- Not so important takes values in the interval] 0,1 .. 0,3]
- Do not care much takes values in the interval [0 .. 0,1]

Using fuzzy values (very important, important, not so important, etc.) facilitates the stakeholders' task of attributing priorities to conflicting aspects. Therefore, for viewpoint Gizmo, for example, Response Time has higher priority than Correctness and Multiple Access, and Correctness has higher priority than Multiple Access.

Resolve conflicts. The conflict mentioned above should not be too difficult to resolve, as the weights express priorities. However, Toll Gate and its sub-viewpoints: Paying Toll (with sub-viewpoints: Single Toll, Exit Toll) and Entry Toll still show a conflicting situation between Response Time and Correctness. These two aspects contribute negatively to each other and have the same weight allocated to them (see the cells highlighted in Table 3). Using Arcade we can determine where that conflict exists (in this case, in Paying Toll). On one hand, the toll gate needs to react in time; on the other hand, it needs to display the correct amount. To resolve these kinds of

conflicting situations negotiation among stakeholders is needed. One suitable solution will be to lower the weight allocated to Response Time to 0.8 for the affected viewpoints. This is because Correctness is more important than Response Time. It is essential that the correct amount is displayed (and subsequently billed) even though the driver may not see it (if s/he is driving too fast).

Table 3. Matrix with weights to conflicting aspects

(**P:** Police; **Gz:** Gizmo; **DS:** Debiting System; **TG:** Toll Gate; **PT:** Paying Toll; **ST:** Single Toll; **ExT:** Exit Toll; **ET:** Entry Toll; **Vh:** Vehicle; **UV:** Unauthorized Vehicle; **VO:** Vehicle Owner; **Act:** Activation; **Reg:** Registration; **Bill:** Billing; **Adm:** Administration)

VP / Aspects	P	Gz	DS	ATM	TG	PT	ST	ExT	ET	Vh	UV	VO	Reg.	Act.	Bill.	Adm..
Response Time		*1,0*		*0,3*	*1,0*	*1,0*	*1,0*	*1,0*	*1,0*	*1,0*	*1,0*					
Availability		✓		✓	✓	✓	✓	✓	✓				✓	✓		✓
Security	✓		✓	*1,0*									✓		✓	✓
Legal Issues	✓							✓							✓	
Compatibility	✓		✓	✓										✓		
Correctness	✓	*0,8*	✓		*1,0*	*1,0*	*1,0*	*1,0*	*1,0*			✓	✓	✓	✓	
Multi Access		*0,3*		*0,3*	*0,3*	*0,3*	*0,3*	*0,3*	*0,3*	*0,3*	*0,3*		✓	✓		✓

Once all the conflicts have been resolved the specification is revised and recomposition carried out to identify any further conflicts.

3.3.7 Specify Aspect Dimensions

Specification of a candidate aspect's dimensions makes it possible to determine its influence on later development stages and identify its mapping onto a function, decision or aspect, if aspect-orientation is used later. However, this development strategy is not imposed. Indeed, one could use frameworks or patterns to implement an aspect, for example. Patterns, for instance, are interesting from another perspective as well: they help identifying state-dependent aspects. These are better seen when the state design pattern is used to show crosscutting states.

Going back to our example, consider our Compatibility candidate aspect. The requirements derived from this aspect will influence parts of the system specification, architecture and design pertaining to requirements derived from viewpoints constrained by it. They will also influence system evolution as change of the user's ATM cards must be anticipated. The Compatibility aspect will, however, map on to a function allowing activation and reactivation of the gizmo. The Response Time aspect, on the other hand, will influence the type of architecture chosen and the design of the classes realising the requirements constrained by Response Time. It will map to an aspect at the design and implementation level because response time properties cannot be encapsulated in a single class and will be otherwise spread across a number of classes.

4 Modelling Aspects Using a Transformation Approach (MATA)

MATA [27] is an aspect-oriented modelling tool that considers aspect composition as a special case of model transformation. In MATA, the joinpoint model is defined by a

diagram pattern which allows for very expressive joinpoints. For example, a joinpoint may define a sequence of messages. This is in contrast to most previous approaches to aspect-oriented modelling that only allow joinpoints to be single model elements, such as a single message. Also, MATA supports more expressive composition types. For example, an aspect sequence diagram can be composed with a base sequence diagram, using parallel, alternative or loop fragments as part of the composition rule. Most other approaches have often been limited to the before, after, around advice of AspectJ.

4.1 MATA Basic Principles

The composition mechanism of MATA is based on graph transformations. A graph transformation is a graph rule r: L → R from a left-hand side (LHS) graph L to a right-hand side (RHS) graph R. In MATA the composition of a base model, Mb, with an aspect model, Ma, which crosscuts the base, is specified by a graph rule, r: LHS →RHS:

- A pattern is defined on the left-hand side (LHS), capturing the set of points in Mb where new model elements should be added;
- The right-hand side (RHS) defines those new elements and specifies how they should be added to Mb.

MATA supports composition for several UML diagrams (e.g., class, sequence, activity and state diagrams). It represents graph rules in UML's concrete syntax, with some extensions to allow for more expressive pointcut and variable expressions. A MATA rule is given in a diagram, by using the following stereotypes:

- «create»: applied to any model element, specifying the creation of an element.
- «delete»: applied to any model element, specifying the deletion of an element.
- «context»: used with container elements that are created; it avoids creating an element inside a created element, forcing it to match an element in the base.

Figure 7 shows two examples of MATA rules defined in the context of sequence diagrams. R1 specifies that the aspectual behaviour consists of an interaction between 2 objects that must be instantiated to 2 objects in the base. The rule says that the fragment par (that specifies parallelism) and messages r and s in one of the sections of the fragment are created, i.e., they define the aspectual behaviour that must be inserted in the base. However, since p is defined as <<context >>, it must be matched against a message with the same name in the base. The resulting composed model when applying R1 is shown on the top right-hand corner of the figure. Note that since q and b are not part of the rule they come after the par fragment.

Rule R2 is similar, the main difference is the use of the "any" operator. This allows that, in the example, any sequence of messages between p and q can happen in the base (in this case, only the q message).

4.2 Scenario Modelling with Aspects: Illustrating with a Case Study

We first show the application of MATA when modelling aspectual scenarios, i.e., scenarios that crosscut other scenarios. To model aspectual scenarios we first identify use cases. Then, for each use case we identify the possible scenarios (main and secondary). Then, by analysing them we find the crosscutting behaviour (or aspectual scenarios). Next, we model in detail the base scenarios using sequence diagrams and the aspectual scenarios using MATA which encapsulates the composition rule.

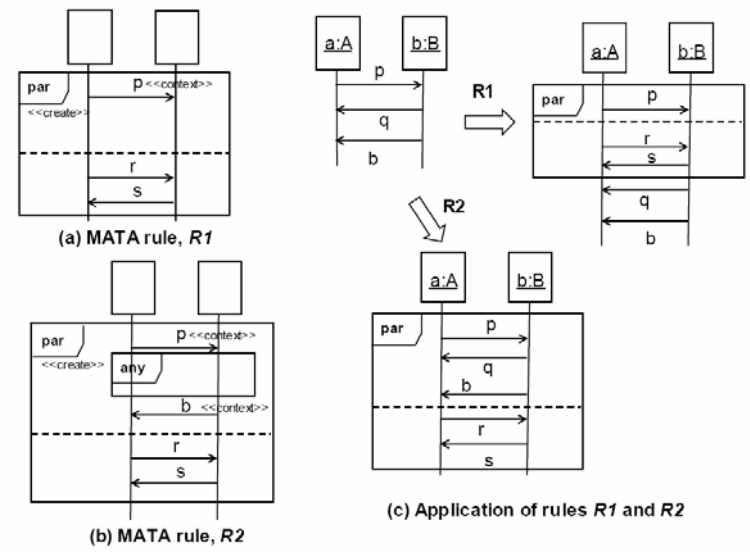


Fig. 7. MATA rules [2]

We will illustrate this using a simple car parking example [1]. The top-level requirements for the car parking system are as follows: "To use a car parking system, a client has to get a ticket from a machine after pressing a button. Afterwards, the car is allowed to enter and park in an available place. The system has to control if the car parking is full or if it still has places left. When s/he wants to leave the parking place, s/he has to pay the ticket obtained (described above) in a paying machine. The amount depends on the time spent. After paying the client can leave by inserting the ticket in a machine which will open the gate for her/him to leave. Regular users of the parking system may pre-purchase time and enter/exit by inserting a card and PIN number which will result in money being deducted automatically from the user's account."

4.2.1 Identify Use Cases, Aspectual and Non-aspectual Scenarios

From the requirements above, we identify the use cases Enter Lot, Exit Lot and Pay. Figure 8 shows a use case diagram.

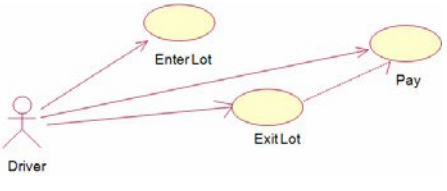


Fig. 8. Use Case Diagram for the Car Parking System

Next, we refine each use case into scenarios. Table 4 shows a non-exhaustive list of scenarios. Among these, some crosscut others. For example, handling error situations are tangled with normal ones. Also, these handling error situations crosscut several scenarios. This is the case of broken machinery, incorrect PIN, etc.

Table 4. Car parking scenarios

S1	Enter Lot, parking lot has space
S2	Enter Lot, parking lot has no space
S3	Enter Lot, regular user types in PIN and enters
S4	Enter Lot, regular user types in PIN; PIN incorrect
S5	Enter Lot, parking lot has space; machine is broken
S6	Exit Lot, driver inserts ticket; ticket paid
S7	Exit Lot, driver inserts ticket; ticket not paid
S8	Exit Lot, driver has no ticket
S9	Exit Lot, grace period from paying ticket exceeded
S10	Exit Lot, regular user types in PIN and exits
S11	Exit Lot, driver types in PIN; insufficient funds in account
S12	Exit Lot, driver inserts ticket; machine is broken
S13	Exit Lot, driver inserts ticket; ticket cannot be read
S14	Exit Lot, driver types in PIN; PIN incorrect
S15	Pay, driver inserts ticket, correct money inserted
S16	Pay, driver inserts ticket; ticket cannot be read
S17	Pay, driver inserts ticket; machine is broken
S18	Pay, driver adds money to PIN card

This leads to the scenarios given in Tables 5 and 6, where I1-I11 are non-aspectual and A1-A3 are aspectual. For example, A1 is an aspectual scenario as it crosscuts the non-aspectual scenarios I3, I4, and I10.

Table 5. Non-Aspectual Scenarios

I1	Enter Lot, parking lot has space
I2	Enter Lot, parking lot has no space
I3	Enter Lot, regular user types in PIN and enters
I4	Exit Lot, driver inserts ticket; ticket paid
I5	Exit Lot, driver inserts ticket; ticket not paid
I6	Exit Lot, driver has no ticket
I7	Exit Lot, grace period from paying ticket exceeded
I8	Exit Lot, regular user types in PIN and exits
I9	Exit Lot, driver types in PIN but insufficient funds in account
I10	Pay, driver inserts ticket and correct money
I11	Pay, driver adds money to PIN card

Table 6. Aspectual Scenarios

A1	Machine is broken
A2	Ticket cannot be read
A3	PIN incorrect

4.2.2 Describe Aspectual and Non-aspectual Scenarios

Figure 9 shows the MATA sequence diagram for interaction aspect A1. If the machine cannot respond the supervisor is alerted and the driver receives an error message. The diagram contains three role names that must be instantiated to compose the aspect with UML sequence diagrams.

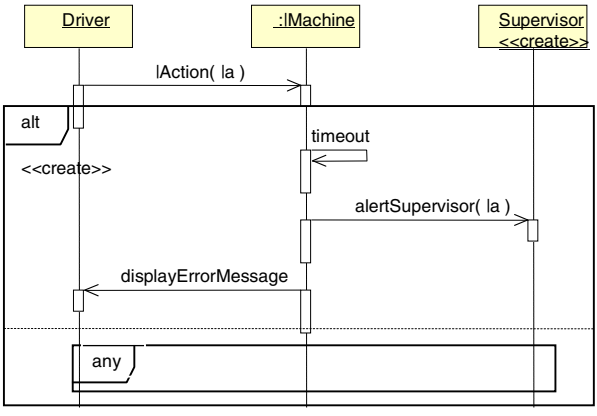


Fig. 9. The aspectual scenario "Machine is broken"

The non-aspectual scenario I4 is depicted in Figure 10. Having the ticket inserted, the Lot Exit Machine checks it and if it is valid the transaction is recorded. Then the ticket is ejected and the barrier opens. Once the driver gets the ticket and leaves, the barrier closes.

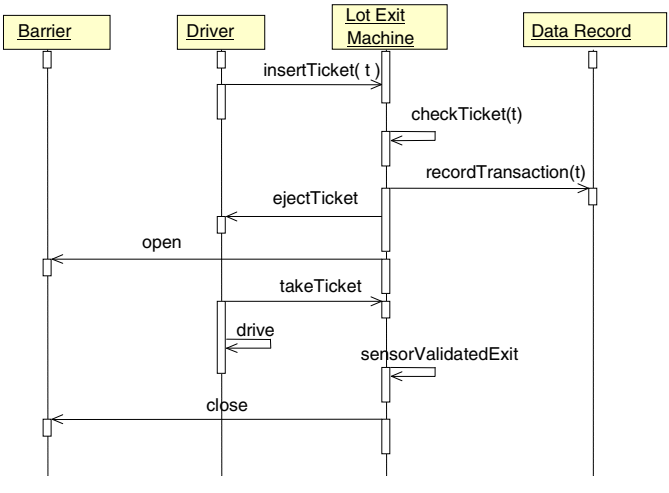


Fig. 10. Sequence diagram for exiting with paid ticket

The composed scenario is shown in Figure 11. First we instantiate the roles: IMachine binds to Lot exit Machine, IAction binds to insertTicket(t) and Ia binds to t. Then we compose, based on the graph transformations mechanisms of MATA.

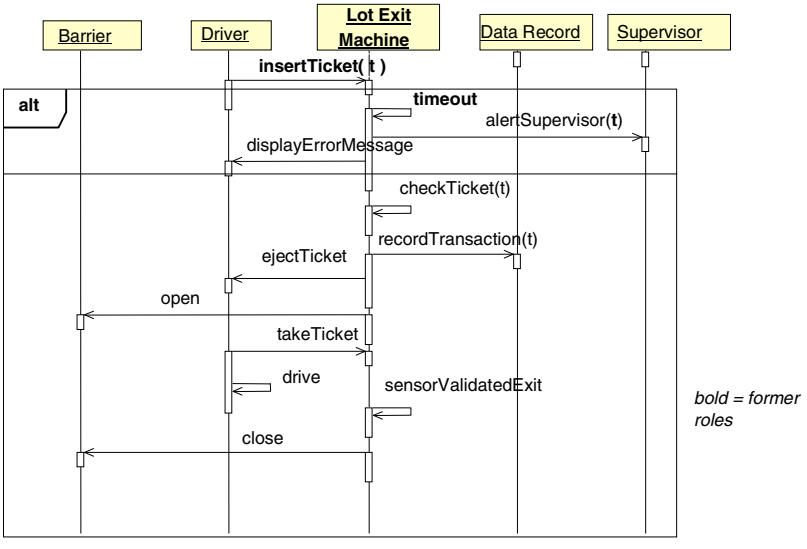


Fig. 11. Composed scenario

4.3 Specifying Volatile Behaviour as Aspects

Volatile requirements are business rules that are prone to change during the software life. Thus, we would like to be able to change them quickly, at any time. Our proposal is to handle volatility as aspects, since both concepts share independency, modular representation and composition with a base description. Doing so, volatility is modularized and requirements modifications can be rapidly instantiated and composed into an existing system.

Figure 12 shows a model to handle volatile requirements [16]. The process starts identifying the problem domain concerns and follows by classifying them either as a service or a constraint and either enduring or volatile. Each concern is described in terms of its main elements in a template illustrated in Tables 7 and 8. Refactoring may be needed if a concern description, for example, is too complex, justifying the decomposition of this concern into two or more sub-concerns (step 3).

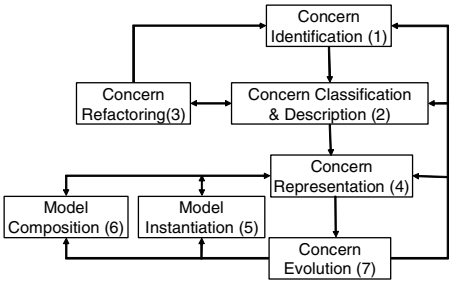


Fig. 12. Aspect-oriented evolutionary model for volatile concerns [16]

Concerns are represented using UML or MATA diagrams. Enduring services are modelled using UML. During this task, crosscutting elements in a model, or crosscutting models can be identified. Volatile concerns, crosscutting concerns and constraints are defined as role elements in the representation models. Representing these concerns as roles requires that the original concern definition is modified to become non-specific, thus allowing several concrete instantiations (step 5).

Concern evolution (step 7) may require that new concerns be identified, classified, refactored and modelled, iteratively. At this stage, the outcome of the process is a specification where core concerns and concern roles are kept separate. Instantiation and/or composition (steps 5 and 6) can take place at the level of granularity of elementary concerns or models. While instantiation offers the opportunity to make concrete decisions regarding volatile concerns, which have been marked as role elements, composition serves to weave the instantiated concerns into a base model that contains enduring services. Composition is facilitated using MATA.

The process just described is now illustrated using an automated transport system[2] in which "transport contractors bid to fulfil passenger transport orders. Passenger orders can be bid on by all transport contractors and the lowest bid wins. In the case of two lowest bids, the first arriving bid wins. Successful completion of an order results in a monetary reward for the shuttle involved. In case an order has not been completed in a given amount of time, a penalty is incurred."

4.3.1 Concern Identification, Classification and Description

The identification of concerns involves the identification of stakeholders, analysing documents that describe the problem, reusing catalogues [5], stakeholders' interviews transcripts, etc. In our example, two concerns are identified: (C1) Passenger orders can be bid for by all transport contractors and the lowest bid wins. In the event of two lowest bids, the first arriving bid wins. (C2) Successful completion of an order results in a monetary reward for the shuttle involved. In case an order has not been completed in a given amount of time, a penalty is incurred.

Concerns are classified according to their type, i.e., enduring, volatile, services or constraints. For example, concern C1 is a service that might be classified as both enduring and volatile. While the first sentence refers to something stable as it is likely that shuttles will always have to bid for business in this system, the second implies a choice process which is likely to change depending on organization policies. This leads to a natural refactoring of this concern into two separate concerns—one to capture the enduring part and one to capture the volatile part.

Each concern is detailed using a template that collects its contextual and internal information. Tables 7 and 8 illustrate the templates for concern C1 (refactored into C1a and C1b). The "Interrelationships" lists the concerns that a given concern relates to. A responsibility is an obligation to perform a task.

4.3.2 Concern Refactoring

In the transport system example, the concern "(C1) Passenger orders can be bid for by all transport contractors and the lowest bid wins. In the event of two lowest bids, the first arriving bid wins." could be decomposed into two separate concerns—one for the

[2] Shuttle system description found at `http://scesm04.upb.de/case-study-1/ShuttleSystem-CaseStudy-V1.0.pdf`

bidding (C1a) and one for the decision on who wins in the event of two equal lowest bids (C1b). Identified volatile concerns may be redefined to represent a more generic concern. For example, C1b if originally defined as *Choosing From Equal Bids*, can be generalized to *Choose Bid*. Such a generalization facilitates software change since a designer may want to change the bidding policies in the future.

The classification process helps to refactor the list of concerns into a list with consistent granularity. This is because increased granularity is often needed to be able to specify the fact that part of a concern is enduring or volatile. As an example, for concern (C1) above, one would like to say that the first part of the concern (the bidding process) is enduring, whereas the second part (dealing with two lowest bids) is volatile—one might, for example, later wish to use a different selection strategy in which bidders with strong performance histories win equal bids. Such a classification would lead naturally to splitting concern (C1) into two concerns (C1a) and (C1b). Applying a classification strategy consistently across a set of concerns leads to a consistent level of granularity in concern representation.

Table 7. *Order Handling* description

Concern #	C1a
Name	Order Handling
Classification	Enduring service
Stakeholders	Shuttle, Passenger
Interrelationships	C1b, C2
List of pre-conditions	
(1) There is a new order	
List of responsibilities	
(1) Broadcast order (2) Receive bids (3) Store bids	

Table 8. *Choose Bid* description

Concern #	C1b
Name	Choose Bid
Classification	Volatile service
Stakeholders	Shuttle
Interrelationships	C1a
List of pre-conditions	
(1) There should be at least one order	
List of responsibilities	
(1) Get offers (2) Select winning bid (3) Store Choice (4) Make decision known	

4.3.3 Concern Representation

Concerns are represented using UML use case and activity models. Elements in a model representing crosscutting concerns or volatile constraints and services are marked as roles and the model becomes a pattern modelled using MATA notation.

Build use case models. A MATA use case model is a modified use case model with use case roles, each one representing volatile constraints and services. It incorporates use case roles, where concerns are mapped into use cases, volatile constraints and services are mapped into use case roles, stakeholders are mapped into actors and interrelationships help in identifying relationships between use cases. Apart from the <<include>> and <<extend>> relationships we also have those that are derived from constraints, the new relationship <<constrain>>, meaning that the original use case restricts the behaviour of the destination use case. Some of the use cases derived from constraint concerns are typically global properties, such as NFRs. Figure 13 illustrates a MATA use case model for the transport system, where C1a (in Table 7) and C1b (in Table 8) are represented by use cases. Note how C1b is given as a role use case, pointing out the clear distinction between enduring and volatile concerns—a reader of the model can immediately see where the volatility lies.

Identify crosscutting concerns. Crosscutting concerns are those that are required by several other concerns. This can be found in the concerns' templates, or looking at the relationships between use cases in the MATA use case model. For example, one use case that is included by or extends several other use cases is crosscutting.

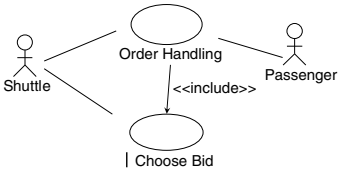
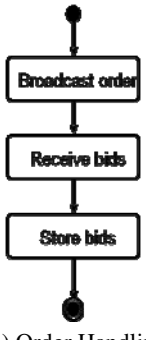
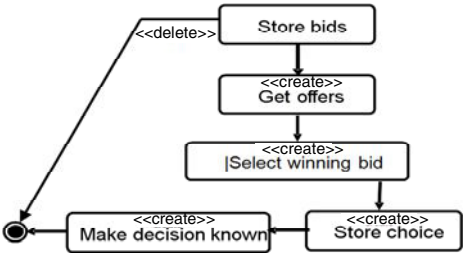


Fig. 13. Transport UCPS

Build activity models. These describe the use cases' behaviour. Each responsibility in the concern's template corresponds to an activity in an activity diagram or an activity role in a MATA activity diagram. The nature of the concern (crosscutting, enduring or volatile) decides whether activities or activity roles are used. For example, C1b is volatile; therefore, one or more of its responsibilities will correspond to activity roles in the activity diagram. Activity roles are those that correspond to the responsibilities that are primarily responsible for making the concern volatile. In this case, responsibility 2 of C1b will correspond to a role activity (Figure 14.b).



(a) Order Handling



(b) Choose Bid

Fig. 14.

Model instantiation. Model elements can be instantiated by a rule of the form:

```
<step #.> Replace |<modelElement A> with <modelElement B>
```

This means that *modelElement A* is eliminated and substituted by *modelElementB*, including its context. For example, consider the concern C1b, represented in the MATA use case model as `|Choose Bid`. The instantiation rule is as follows:

```
1. Replace |Choose Bid
     with Choose From Bids (Equal Bids Choice Based On Arrival Time)
```

An instantiation for MATA activity model in Figure 14 (b) is:

```
2. Replace |Select Winning Bid
     with Select Lowest Bid (Equal Bids Choice Based On Arrival Time)
```

Model composition. Composition and instantiation can be applied independently from each other in an incremental fashion, leading to consecutive refinements of abstract requirements models into more concrete analysis models. In a more traditional aspect-oriented view, only crosscutting concerns would be composed with base modules. Here, we use composition to weave aspectual or volatile models to base models. As discussed before, the composition rule is specified in MATA diagrams. In the example, Figure 14(b) specifies that the Store bids activity must exist in the base model (Figure 14 (a)) and that the transition from Store bids to the final state will be deleted. The resulting model is illustrated in Figure 15.

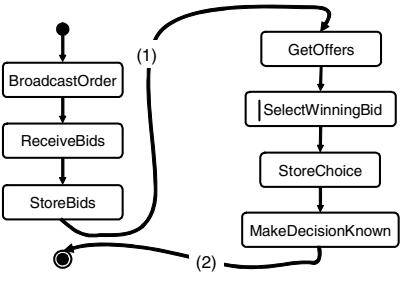


Fig. 15. Resulting composed model

In this particular case, the choice of a particular method for choosing the winning bid would be performed after this composition. When the requirements change, composition can be used to update the model in a less difficult and modular way.

Concern evolution. Evolution should cope with changes in concerns that are already part of the system and with new functionalities or constraints not yet part of the existing system. In the former, the system is prepared to handle the change, by either defining a new instantiation rule, or else by changing one or more composition rules. For example, a change in the process used to select the winning bid (C1b) is easily handled at all levels by specifying the appropriate instantiations:

```
1. Replace |Choose Bid
       with Choose From Bids (Equal Bids Choice Based On History)
2. Replace |Select Winning Bid
       with Select Lowest Bid (Equal Bids Choice Based On History)
```

In cases where we have to remove a concern, we need to remove all dependencies on this concern from all the composition rules. Coping with new requirements or constraints requires the reapplication of the method to identify the corresponding new concerns. These are integrated with the existing system by adding or changing existing composition rules.

5 Conclusions and Discussion

At the end of the tutorial, participants had a clear understanding of: the role of aspect-oriented software development concepts in requirements engineering; techniques, tools and good practice guidelines for identifying, modelling, composing and analysing crosscutting properties at the requirements-level; how aspect-oriented requirements models and their analysis drive development of solution domain models.

To accomplish this goal, we discussed the fundamental concepts of aspect-orientation and showed how aspect-oriented requirements engineering methods can work in tandem with existing requirements engineering ones. We have done this by presenting two different types of aspect-oriented requirements analysis approaches, illustrating them with case studies and their supporting tools. One of the approaches was Arcade, an extension of a viewpoint-oriented approach, and the other was MATA, which employs use cases and handles volatile concerns as aspects to facilitate the systems' evolution. While the first focuses mainly on aspectual requirements that correspond to quality attributes, the second allows the use of UML, a modelling standard that is well-known by the software engineering community.

Acknowledgements

We express our gratitude to Awais Rashid, Jon Whittle and Pete Sawyer from Lancaster University with whom we collaborate since 2002 and who co-authored the two approaches presented here. This work is also supported by FCT/MCTES.

References

[1] Araújo, J., Whittle, J., Kim, D.: Modeling and Composing Scenario-Based Requirements with Aspects. In: 12th IEEE International Requirements Engineering Conference, Japan, pp. 58–67 (2004)

[2] Araújo, J., Moreira, A., Whittle, J.: Aspect-Oriented RE with Scenarios. Tutorial Presented at 15th RE 2007, India (2007), http://www.re07.org/tutorials/#T3

[3] Baniassad, E., Clarke, S.: Theme: An approach for aspect-oriented analysis and design. In: 26th International Conference on Software Engineering, Scotland, pp. 158–167 (2004)

[4] Brito, I., Vieira, F., Moreira, A., Ribeiro, R.: Handling Conflicts in Aspectual Requirements Compositions. In: Rashid, A., Liu, Y. (eds.) Transactions on Aspect-Oriented Software Development. LNCS, vol. 4620, pp. 144–166. Springer, Heidelberg (2007)

[5] Chung, L., Nixon, B., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer, Dordrecht (2000)

[6] Chitchyan, R., Rashid, A., Sawer, P., Garcia, A., Alarcon, M., Bakker, J., Tekinerdogan, B., Clarke, S., Jackson, A.: Survey of Analysis and Design Approaches, TR, AOSD-Europe-ULANC-9 (2005)

[7] Clark, R., Moreira, A.: Constructing Formal Specifications from Informal Requirements. In: Software Technology and Engineering Practice, pp. 68–75. IEEE Computer Society Press, Los Alamitos (1997)

[8] Dijkstra, E.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)

[9] Finkelstein, A., Sommerville, I.: The Viewpoints FAQ. Software Engineering Journal: Special Issue on Viewpoints for Software Engineering, IEE/BCS 11(1), 2–4 (1996)
[10] Jacobson, I., Chirsterson, M., Jonsson, P., Overgaard, G.: Object-Oriented Software Engineering: a Use Case Driven Approach. Addison-Wesley, Reading (1992)
[11] Jackson, M.: Problem Frames: Analyzing and Structuring Software Development Problems. Addison Wesley, Reading (2000)
[12] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. In: Liu, Y., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
[13] Lamsweerde, A.: Goal-Oriented Requirements Engineering: A Guided Tour. In: 5th International Symposium on RE, pp. 249–261. IEEE CS Press, Los Alamitos (2001)
[14] Moreira, A., Araújo, J., Brito, I.: Crosscutting Quality Attributes for Requirements Engineering. In: 14th ACM International Conference on SEKE, pp. 167–174 (2002)
[15] Moreira, A., Rashid, A., Araújo, J.: Multi-dimensional Separation of Concerns in Requirements Engineering. In: 13th IEEE Requirements Engineering Conference, Paris, France, pp. 285–296 (2005)
[16] Moreira, A., Araújo, J., Whittle, J.: Modeling Volatile Concerns as Aspects. In: Martinez, F.H., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 544–558. Springer, Heidelberg (2006)
[17] Pimentel, A., Gomes, C., Moreira, A., Ribeiro, R.A., Araújo, J.: HAM for architectural choices in software development. In: 14th Congress of APDIO, Book of abstract—IO 2009, Lisbon (September 2009)
[18] HAM tool, `http://ample.holos.pt/pageview.aspx?pageid=78&langid=1#ham`, `http://www.ample-project.net`
[19] Rashid, A., Garcia, A., Moreira, A.: Aspect-Oriented Software Development beyond Programming. Tutorial Presented at ICSE 2006 (2006)
[20] Rashid, A., Moreira, A.: Domain Models are NOT Aspect Free. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 155–169. Springer, Heidelberg (2006)
[21] Rashid, A., Moreira, A., Araújo, J.: Modularisation and Composition of Aspectual Requirements. In: ACM International Conference on AOSD, pp. 11–20 (2003)
[22] Rashid, A., Moreira, A., Tekinerdogan, B.: Editoral Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design. IEE - Proceedings Software 151(4), 153–156 (2004)
[23] Ross, D.T., Goodenough, J.B., Irvine, C.A.: Software Engineering: Process, Principles, and Goals. Computer 8(5), 17–27 (1975)
[24] Sampaio, A., Rashid, A., Chitchyan, R., Rayson, P.: EA-Miner: Towards Automation in Aspect-Oriented Requirements Engineering. Trans. on Aspect-Oriented Software Development 3, 4–39 (2007)
[25] Tarr, P., Ossher, H., Harrison, H., Sutton, J.S.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: 21th IEEE International Conference on Software Engineering, Canada, pp. 107–119 (1999)
[26] Weston, N., Chitchyan, R., Rashid, A.: Formal semantic conflict detection in aspect-oriented requirements. Requirements Engineering 14(4), 247–268 (2009)
[27] Whittle, J., Moreira, A., Araújo, J., Rabbi, R., Jayaraman, P., Elkhodary, A.: An Expressive Aspect Composition Language for UML State Diagrams. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 514–528. Springer, Heidelberg (2007)

Lightweight Language Processing in Kiama

Anthony M. Sloane

Department of Computing, Macquarie University, Sydney, Australia
Anthony.Sloane@mq.edu.au

Abstract. Kiama is a lightweight language processing library for the Scala programming language. It provides Scala programmers with embedded domain-specific languages for attribute grammars and strategy-based term rewriting. This paper provides an introduction to the use of Kiama to solve typical language processing problems by developing analysers and evaluators for a simply-typed lambda calculus. The embeddings of the attribute grammar and rewriting processing paradigms both rely on pattern matching from the base language and each add a simple functional interface that hides details such as attribute caching, circularity checking and strategy representation. The similarities between embeddings for the two processing paradigms show that they have more in common than is usually realised.

1 Introduction

Kiama is a language processing library for the Scala programming language [1, 2]. We are distilling the key ideas of successful processing paradigms from language research and making them available in a lightweight library. In other words, we are embedding the paradigms into a general purpose language. The result is a flexible combination of general programming techniques and high-level abstractions suited to many forms of language processing. At present, we are focused on building traditional language processing applications such as compilers, interpreters, generators and static analysis tools, but in the longer term we believe that these paradigms have much to offer in a more general software engineering setting.

Some of the motivation for Kiama comes from experience building generators for the Eli system [3]. Eli translates high-level specifications of language syntax, semantics and translation into C implementations. Eli successfully combines many off-the-shelf tools and custom-built generators that use a variety of specification languages. However, because of their varying origins, the Eli specification languages are often *ad hoc*, have arbitrary differences and lack features that are commonplace in general purpose languages such as name space control, modularity and parameterisation. In our view, as language processing systems are used to tackle larger tasks and different techniques are combined, these issues become particularly problematic.

The Kiama thesis is that in the language processing domain it is better to start with a modern general purpose language that embodies prevailing wisdom about

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 408–425, 2011.

how to structure, scale and extend applications, than to expect every generator builder to incorporate this wisdom into their own specification languages and tools. The approach of Kiama is therefore to combine proven language processing paradigms into a coherent whole, supported by general facilities from a host language.

Kiama is hosted by the Scala programming language that provides both object-oriented and functional features in a statically-typed combination running on the Java Virtual Machine [1, 2]. Thus, Scala constitutes a powerful base on which to experiment with embedding. At present, Kiama supports two main processing paradigms: *attribute grammars* and *strategy-based term rewriting*. Attribute grammars are particularly suited to expressing computations on fixed tree or graph structures, which is needed for static analysis. Rewriting is ideal for describing computations that transform trees for translation or optimisation.

A notable result from our experience embedding attribute grammars and rewriting in Scala is the high degree to which the power of more complex generator-based systems can be realised with a lightweight embedding. For this domain at least, Scala provides just the right level of expressibility for the notations and flexibility for their implementation. Moreover, the two paradigms are embedded in a very similar way based on Scala's pattern matching constructs. Thus, parallels between the paradigms that were not previously obvious are revealed and the new concepts that must be learned by a programmer are limited.

A full comparison of Kiama with related work is beyond the scope of this paper. Nevertheless, it is important to note that the library has been heavily influenced by existing notations and implementations of both attribute grammars and rewriting. The attribute grammar facilities are modelled on those of the JastAdd system [4]. Kiama's term rewriting library is based on the Stratego language and library [5]. Kiama also shares some characteristics with other embeddings of these paradigms, most notably strategic programming in functional languages in the Strafunski [6] and Scrap Your Boilerplate projects [7].

Kiama is released under the GNU Lesser General Public License. Further information including binary distributions, code, documentation, examples and mailing lists can be found on the project site `http://kiama.googlecode.com`.

Outline

This paper presents an overview of Kiama's current capabilities with a focus on how the main features of the JastAdd and Stratego languages are supported via a lightweight embedding. (More detailed discussion of Kiama's relationship to JastAdd can be found in Sloane et al. [8]).

We proceed by developing implementations of typical processing tasks for a simply-typed version of the lambda calculus. This source language was chosen to be familiar and relatively simple, yet to provide processing tasks that are also relevant to other more complex languages.

Section 2 describes the version of lambda calculus used in the rest of the paper and how programs are represented as Scala data structures. Section 3 shows how Kiama's attribute grammar facilities can be used to define static

program analyses. Section 4 implements various forms of evaluation mechanism as rewriting strategies. The paper concludes with a short discussion of Kiama's capabilities and future plans.

The paper presents the main code fragments necessary to achieve the desired effects. No knowledge of Scala is assumed, but experience with object-oriented programming and pattern-matching as in functional languages will be useful. We omit uninteresting scaffolding code that is necessary to turn these fragments into compilable Scala code. The complete source code can be found in the lambda2 example in the Kiama distribution.

2 A Typed Lambda Calculus

To keep things simple but realistic, we use a simply typed lambda calculus as the source language for the processing described in this paper. Figure 1 summarises the abstract syntax of the language.

The Scala version of the abstract syntax is a straight-forward encoding of the abstract syntax using Scala *case classes* (Figure 2). For most purposes, case classes operate as regular classes but also provide special construction syntax and pattern matching support similar to that provided for algebraic data types in functional languages. *Case objects* are the sole instances of anonymous singleton case classes.

As an example of construction, a tree fragment representing the lambda calculus expression

$$(\lambda x : Int\ .\ (\lambda y : Int\ .\ x + y - 2))\ 3\ 4$$

can be constructed in Scala by the expression

```
App (App (Lam ("x", IntType,
                Lam ("y", IntType,
                     Opn (SubOp,
                          Opn (AddOp, Var ("x"),
                                      Var ("y"))),
                          Num (2)))
          Num (3)),
      Num (4))
```

In later sections, we will pattern match against "constructors" such as `Lam` and `App` to deconstruct such expressions.

3 Attribution

Attribute grammars have been widely studied as a specification technique for describing computations on trees [9, 10]. In an attribute grammar, the context-free grammar of a language is augmented with *attribute equations* which define the values of attributes of tree nodes. In their purest form, attribute grammars have no notion of tree updates, so they are best suited to analysis of fixed structures and attributes can be understood as static properties.

Expressions (e)	n	number
	v	variable
	$\lambda v : t \;.\; e$	lambda abstraction
	$e_1\ e_2$	application
	$e_1\ o\ e_2$	primitive operation
Types (t)	`int`	primitive integer type
	$t_1 \rightarrow t_2$	function type
Operations (o)	$+$	addition
	$-$	subtraction

Fig. 1. Abstract syntax of typed lambda calculus

```
abstract class Exp
case class Num (n : Int) extends Exp
case class Var (i : Idn) extends Exp
case class Lam (n : Idn, t : Type, e : Exp) extends Exp
case class App (e1 : Exp, e2 : Exp) extends Exp
case class Opn (o : Op, e1 : Exp, e2 : Exp) extends Exp

type Idn = String

abstract class Type
case object IntType extends Type
case class FunType (t1 : Type, t2 : Type) extends Type

abstract class Op
case object AddOp extends Op
case object SubOp extends Op
```

Fig. 2. Scala data type to represent the lambda calculus abstract syntax

Attribute grammar evaluation approaches can be divided into two broad categories: those that statically analyse attribute dependencies and those that wait until run-time. Kiama's approach is in the latter category [8]. It uses an evaluation mechanism similar to that apparently first used by Jourdan [11], a variant of which is also used in the JastAdd system [4]. Attributes are computed by functions that dynamically demand the values of any other necessary attributes. Attribute values are cached so that they do not needed to be re-evaluated if they are demanded again.

3.1 Free Variables

As a simple example of using attribution to compute a useful property of a tree, consider *free variable analysis* of lambda calculus expressions. We want to calculate a set of the variables that are not bound in a supplied expression. A typical case-based definition of this analysis is as follows [12].

$$fv(n) = \{\}$$
$$fv(v) = \{v\}$$

$$fv(\lambda v : t \;.\; e) = fv(e) - v$$
$$fv(e_1\ e_2) = fv(e_1) \cup fv(e_2)$$
$$fv(e_1\ o\ e_2) = fv(e_1) \cup fv(e_2)$$

A Kiama version of this analysis uses standard Scala pattern matching to specify the same cases and build a Scala `Set` value (Figure 3). In attribute grammar terminology, `fv` is a *synthesised attribute* because it is defined in terms of attributes of the expression and its children.

In Figure 3, `==>` is a Kiama infix type constructor alias for Scala's generic partial function type. Thus, a function of type `T ==> U` transforms values of type `T` into values of type `U`, but may not be defined at all values of type `T`.

```
val fv : Exp ==> Set[Idn] =
  attr {
    case Num (_)          => Set ()
    case Var (v)          => Set (v)
    case Lam (v, _, e)    => fv (e) -- Set (v)
    case App (e1, e2)     => fv (e1) ++ fv (e2)
    case Opn (_, e1, e2) => fv (e1) ++ fv (e2)
  }
```

Fig. 3. Free variable attribute definition. `Exp ==> Set[Idn]` is the type of a partial function from expressions to sets of variable identifiers. The operators `--` and `++` are set difference and union, respectively. An underscore is a wildcard pattern which matches anything.

The code between braces in Figure 3 is standard Scala syntax for an anonymous pattern-matching partial function. Kiama's `attr` function wraps the pattern matching with the dynamic behaviour of non-circular attributes.

```
def attr[T,U] (f : T ==> U) : T ==> U
```

`attr (f)`, where `f` is a partial function between some types `T` and `U`, behaves just like `f` except that it caches its argument-result pairs and detects when a cycle is entered (i.e., when `f (t)` is requested while evaluating `f (t)`, for some node `t`).

The free variables of an expression `e` can now be referenced via a normal function application `fv (e)` or using Kiama's attribute access operator `->` as `e->fv`. The latter is designed to mimic traditional attribute grammar notations.

3.2 Name and Type Analysis

Free variable analysis is a very simple computation defined by a bottom-up traversal of the expression tree. *Name and type analyses* are examples of more complex processing that must be performed by compilers and many other source code analysis tools. An analysis of names is typically needed before type analysis can be performed. Our aim in this section is to analyse expressions such as

```
val env : Exp ==> List[(Idn,Type)] =
  attr {
    case e =>
      (e.parent) match {
        case null                => List ()
        case p @ Lam (x, t, _) => (x,t) :: p->env
        case p : Exp             => p->env
      }
  }
```

Fig. 4. Definition of environment attribute as list of bound variables and their types. Scala's `match` construct performs pattern matching. A pattern `p @ patt` matches against the pattern `patt` and, if successful, binds `p` to the matched value. A pattern of the form `p : T` succeeds if the value being matched is of type `T`, in which case it binds `p` to the value. An underscore is a pattern that matches anything. `::` is the List prepend operation.

$\lambda x : Int\ .\ (\lambda y : Int \to Int\ .\ y\ x)$ and determine that the application $y\ x$ is legal because y is a function from integer to integer and x is an integer.

It is not immediately obvious how to achieve the appropriate traversals of an expression tree to perform name and type analysis. Do we first perform a traversal for name analysis and then one for type analysis, or can we mix them somehow? The attribute grammar paradigm helps considerably with avoiding these questions because it enables us to concentrate on the dependencies between attributes. Dynamic scheduling of attribute computations will take care of the traversal. Therefore, we do not need to explicitly separate name and type analysis.

An environment-based analysis. First, we present a name and type analysis that uses explicit environment structures to keep track of the bound names and their types. An alternative where the tree itself holds this information is presented in the next section.

The `env` attribute computes an environment for a given expression consisting of all variables that are visible at that expression and their types. The environment is represented by a list, with the interpretation that earlier entries hide later ones, thereby implementing variable shadowing.

In contrast to the `fv` attribute which was defined by matching on the node itself, the `env` attribute is defined by cases on its parent. In other words, the names that are visible at a node depend on the node's context. We have three cases: a) at the top of the tree (null parent) nothing is visible, b) inside a lambda expression, the visible names are whatever is visible at the lambda node plus the name bound at that node, and c) in all other cases, the names visible at a node are just those that are visible at the node's parent. These cases are easily specified by pattern matching (Figure 4). In attribute grammar terms, `env` is an *inherited attribute*.[1]

[1] In some cases, not shown in this overview, it is useful to match on both the node and its parent. Therefore, the synthesised versus inherited distinction is not particularly meaningful in Kiama, since each attribute definition is free to access any part of the tree that it needs.

```
val tipe : Exp ==> Type =
  attr {
    case Num (_) =>
      IntType

    case Lam (_, t, e) =>
      FunType (t, e->tipe)

    case App (e1, e2) =>
      e1->tipe match {
        case FunType (t1, t2) if t1 == e2->tipe =>
          t2
        case FunType (t1, t2) =>
          message (e2, "need " + t1 + ", got " +
                       (e2->tipe))
          IntType
        case _ =>
          message (e1, "application of non-function")
          IntType
      }

    case Opn (op, e1, e2) =>
      if (e1->tipe != IntType)
        message (e1, "need Int, got " + (e1->tipe))
      if (e2->tipe != IntType)
        message (e2, "need Int, got " + (e2->tipe))
      IntType

    case e @ Var (x) =>
      (e->env).find { case (y,_) => x == y } match {
        case Some ((_, t)) => t
        case None =>
          message (e, "'" + x + "' unknown")
          IntType
      }
  }
```

Fig. 5. Definition of the expression type attribute. Kiama's `message` operation records a message associated with a particular tree node. Scala's `find` method searches a list using the predicate provided as an argument and returns an `Option[T]` value, where `T` is the list element type. A value of type `Option[T]` is either `Some (t)` for some value `t` of type `T`, or it is `None`.

Kiama provides fields called *structural properties* that give generic access to the tree structure. For example, the `parent` field of `e` used in Figure 4 is a structural property that provides access to the parent of any node, or null if the node is the root. The other structural properties provided by Kiama are `isRoot` for all nodes, and `prev`, `next`, `isFirst` and `isLast` for nodes occurring in sequences. The structural properties are provided automatically to any class that inherits Kiama's `Attributable` trait, as in

```
abstract class Exp extends Attributable
```

With `env` in hand, we can define the `tipe` attribute[2] that gives the type of any expression (Figure 5). Unlike traditional typing rules, the environment is not passed, because it can be accessed directly using the `env` attribute as needed. There are five cases:

a) a number has integer type,
b) a lambda expression $\lambda x : t \;.\; e$ has type $t \to t_e$ where t_e is the type of e,
c) an application of a function of type $t_1 \to t_2$ to an expression of type t_1 is of type t_2,
d) the operands and result of an operation are integers, and
e) the type of a variable is the type associated with that variable name in the environment.

If none of these cases apply, a typing error is reported using Kiama's message facility and an error type of `IntType` is returned. (Of course, this approach may lead to spurious errors. A more robust implementation would return a dedicated error type and ensure that values of the error type were acceptable in any context.)

A reference-based analysis. In the environment-based analysis, we reuse the type nodes of the tree when constructing the environment (in the `Lam` case). We can go further and do away with the environment completely by observing that each binding can be represented by the lambda expression in which it is created. This kind of observation is at the heart of Hedin's Reference Attribute Grammars [13], which can be achieved in Kiama with the facilities we have seen already.

All of the cases for the `tipe` attribute stay the same as in the environment version, except for the variable case (Figure 6). Instead of looking up the name in the environment, we define a `lookup` attribute that traverses the tree to find the name if it can. There are three cases: a) we are examining a lambda expression that defines the name we are looking for, so return that lambda expression, b) we are at the root of the tree, so report that we didn't find a binder for the name, and c) ask the parent to lookup the name. `lookup` is therefore a *parameterised, reference attribute* in attribute grammar terminology.[3] In the variable case of `tipe` we can now use `lookup` to find the binder of the name, if there is one.

In `lookup`, the parent reference `e.parent[Exp]` requires the type annotation `Exp` because `parent` is generic and the compiler is not able to infer that expressions only ever occur as children of expressions. The necessity for this annotation reveals a limitation in the embedding approach due to full grammar knowledge not being available when the abstract syntax is implemented as a class hierarchy. The type annotation results in a cast to the given type and it is up to the developer to ensure that the cast cannot fail. More discussion of this issue can be found in Sloane et al. [8].

[2] `type` cannot be used since it is a Scala keyword.
[3] This use of a parameterised attribute defined by a Scala function is simple, but it may not provide the desired caching behaviour, since the parameter value is not included in the cache key. Kiama also provides a variant of `attr` that can be used if more advanced caching is important.

```
def lookup (name : Idn) : Exp ==> Option[Lam] =
  attr {
    case e @ Lam (x, t, _) if x == name =>
      Some (e)
    case e if e.isRoot =>
      None
    case e =>
      e.parent[Exp]->lookup (name)
  }

val tipe : Exp ==> Type =
  attr {

    ...

    case e @ Var (x) =>
      (e->lookup (x)) match {
        case Some (Lam (_, t, _)) =>
          t
        case None =>
          message (e, "'" + x + "' unknown")
          IntType
      }

  }
```

Fig. 6. Definition of the name lookup attribute and the new case for name and type analysis of variables

4 Rewriting

Kiama's rewriting library is closely modelled on the Stratego rewriting language [5]. Stratego uses a general notion of a *rewriting strategy* that takes as input a term representing a tree structure, and either succeeds, producing a (possibly) rewritten term, or fails. Stratego has a rich language of strategy combinators and library strategies that achieve choice, iteration and other more complex term traversal patterns.

The aim for this part of Kiama was to see how much of Stratego could be realised using a pure embedding approach, in contrast to the standard implementation which compiles to C. As this section shows, most of Stratego can be easily encoded. Our encoding is based around a functional abstraction similar to that used for attribute equations in the previous section. Standard Scala pattern matching can be used within rewrite rules. Implementations of the Stratego combinators and library strategies enable most Stratego programs to be written using almost the same syntax.

This section presents examples of using Kiama's rewriting library to evaluate lambda calculus expressions, based on Stratego versions of the same [14].

```
val s =
  reduce (beta + arithop)

val beta =
  rule {
    case App (Lam (x, _, e1), e2) =>
      substitute (x, e2, e1)
  }

val arithop =
  rule {
    case Opn (op, Num (l), Num (r)) =>
      Num (op.eval (l, r))
  }
```

Fig. 7. Definition of simple reduction strategies. The function `substitute` is assumed to implement capture-free substitution. Each primitive operator is assumed to have a method `eval` that evaluates that operator on two integers and returns the result.

4.1 Evaluation

The evaluation strategies fit into a general framework. The interface to an evaluator is a function `eval` that takes an expression and returns the expression that is the result of evaluation.

```
def eval (exp : Exp) : Exp =
  rewrite (s) (exp)

val s : Strategy
```

Evaluation is achieved by rewriting with the strategy `s` which is defined in various ways in the following sections. `rewrite` applies its strategy argument to its term argument. If the strategy succeeds, `rewrite` returns the resulting term, otherwise, it returns the original argument.

4.2 Basic Reduction

The basic evaluation rule for lambda calculus is *beta reduction* [12].

$$(\lambda x : t \;.\; e_1)\; e_2 \rightarrow [e_2/x]e_1$$

where $[e_2/x]e_1$ means capture-free substitution of e_2 for occurrences of the variable x in e_1. Primitive operations can be evaluated by reduction rules that use operations in the meta-language.

Figure 7 shows an encoding of these rules as strategies in Kiama. Each rule is written as a pattern matching function on the relevant tree structure.[4] The pattern match is wrapped by a call to Kiama's `rule` function that converts the function into a strategy.

```
def rule (f : Term ==> Term) : Strategy
```

[4] While the different rules could be combined into a single one, we prefer to keep them separate to enable more flexible reuse.

A `Strategy` is a function from `Term` to `Option[Term]`. The `Option` wrapper is used to represent success and failure. `rule` lifts a partial function `f` to the `Strategy` type, mapping undefinedness of `f` to `None`, representing failure of the strategy. In other words, `rule (f)`, for some partial function `f`, when applied to a term `t`, succeeds with the result of `f (t)`, if `f` is defined at `t`, otherwise it fails.

The `beta` and `arithop` strategies are combined in Figure 7 to form `s` using the non-deterministic choice operator `+`. Finally, the library strategy `reduce` is used to repeatedly apply the basic strategies to the subject term until a fixed point is reached.

4.3 The `Reduce` Strategy

The definition of `reduce` shows both the power of the Stratego language for combining strategies, but also the relatively clean way that this language can be embedded into Scala. In Stratego, `reduce` is defined in terms of other library strategies and basic combinators as

```
try (s)    = s <+ id
repeat (s) = try (s; repeat (s))
reduce (s) = repeat (rec x (some (x) + s))
```

where the new Stratego constructs are

- the identity strategy (`id`) which always succeeds without changing the subject term,
- deterministic choice (`<+`) where the second strategy is only applied to the subject term if the first strategy fails,
- sequential composition (`;`), where the second strategy is applied to the result of a successful invocation of the first,
- definition of a locally recursive binding of `x` by `rec x`, and
- the primitive traversal combinator `some` whose result succeeds if the argument strategy succeeds on at least one child of the subject term.[5]

Thus, we can see that `try` attempts to apply its argument strategy but leaves the term unchanged if that strategy fails. `repeat` applies a strategy repeatedly until it fails. `reduce` repeatedly applies a strategy to sub-terms and the subject term itself until all of those applications fail, upon which it succeeds with the most recent result.

Stratego programs are built up in this way from a collection of primitives and a large library. The result is an extremely expressive language of tree traversal and transformation. Similar power can be achieved in Kiama using notations that are very similar to those of Stratego, even though we rely entirely on Scala syntax and concepts.

Figure 8 shows the Kiama version of the library strategies needed for the basic reduction example. Scala's ability to define methods with symbolic names means

[5] Stratego and Kiama also have `all` and `one` that require success on all children or one child, respectively.

```
def attempt (s : => Strategy) : Strategy =
  s <+ id

def repeat (s : => Strategy) : Strategy =
  attempt (s <* repeat (s))

def reduce (s : => Strategy) : Strategy = {
  def x : Strategy = some (x) + s
  repeat (x)
}
```

Fig. 8. Kiama version of Stratego library combinators. A parameter type preceded by `=>` indicates a pass-by-name mode.

that the primitive combinators `<+` and `+` can be provided as `Strategy` methods; we use `<*` for sequencing, since semicolon is already claimed for other purposes by the Scala syntax. Similarly, `try` is renamed `attempt` since the former is a Scala keyword. Other than these cosmetic changes, the main differences between the two versions are the inclusion of the type information and a more verbose definition for the recursive value `x` in `reduce`.

4.4 Explicit Substitution

Instead of relying on a separate `substitute` function to implement the core of the beta reduction rule, explicit substitutions can be used to bring the entire evaluation process into the rewriting paradigm.

Figure 9 shows how an explicit substitution version can be written in Kiama. First, a new `Let` tree construct is declared to represent substitutions. The substitution $[e_2/x]e_1$, where x has type t, will be represented by the expression `Let ("x", t, e2, e1)`

`s` is now defined in terms of a `lambda` strategy which in turn combines beta reduction (modified to produce an explicit substitution), primitive evaluation (unchanged) and a set of new strategies that implement substitution. `subsVar` actually performs substitution on a variable reference, whereas the others propagate substitutions inward. (As before, a single rule could be used instead of these reusable pieces.)

4.5 Innermost Evaluation

Using `reduce` has an efficiency penalty because it repeats its search for a reducible expression starting from the top of the whole expression each time. An *innermost evaluation* reduces sub-terms before trying to reduce the subject term. Stratego's `innermost` library strategy is defined as follows in terms of a more general `bottomup` traversal strategy.

```
innermost (s) = bottomup (try (s; innermost (s)))
bottomup (s)  = all (bottomup (s)); s
```

both of which are defined in an analogous way in the Kiama library.

```
case class Let (name : Idn, tipe : Type, exp : Exp,
                body : Exp) extends Exp

val s =
  reduce (lambda)

val lambda =
  beta + arithop + subsNum + subsVar + subsApp +
  subsLam + subsOpn

val beta =
  rule {
    case App (Lam (x, t, e1), e2) =>
      Let (x, t, e2, e1)
  }

val subsNum =
  rule {
    case Let (_, _, _, e : Num) => e
  }

val subsVar =
  rule {
    case Let (x, _, e, Var (y)) if x == y => e
    case Let (_, _, _, v : Var)           => v
  }

val subsApp =
  rule {
    case Let (x, t, e, App (e1, e2)) =>
      App (Let (x, t, e, e1), Let (x, t, e, e2))
  }

val subsLam =
  rule {
    case Let (x, t1, e1, Lam (y, t2, e2)) if x == y =>
       Lam (y, t2, e2)
    case Let (x, t1, e1, Lam (y, t2, e2)) =>
      val z = freshvar ()
      Lam (z, t2, Let (x, t1, e1,
                       Let (y, t2, Var (z),
                            e2)))
  }

val subsOpn =
  rule {
    case Let (x, t, e1, Opn (op, e2, e3)) =>
      Opn (op, Let (x, t, e1, e2), Let (x, t, e1, e3))
  }
```

Fig. 9. Definition of reduction with explicit substitutions. In `subsLam`, `freshvar` is a helper function that returns a unique variable name each time it is called.

`innermost` can be used with the `lambda` strategy defined in the previous section to achieve a more efficient evaluation. In Kiama syntax, we have

```
val s = innermost (lambda)
```

4.6 Eager Evaluation

An innermost evaluation is still not very realistic since in a programming language implementation based on lambda calculus it is unlikely that reductions will be performed inside the body of a lambda expression until that expression is applied to an argument. *Eager evaluation* reduces the arguments of applications before the reduction of applications.

An evaluation strategy to express this pattern of evaluation is as follows.

```
val s : Strategy =
  attempt (traverse) <* attempt (lambda <* s)
```

First, we traverse the expression to evaluate any parts of it that should be evaluated before reduction at the top-level of the expression is attempted. The `lambda` strategy from earlier can be reused and augmented with a simple traversal strategy that controls exactly which sub-terms are reduced first.

```
val traverse : Strategy =
  rule {
    case App (e1, e2) =>
      App (eval (e1), eval (e2))
    case Let (x, t, e1, e2) =>
      Let (x, t, eval (e1), eval (e2))
    case Opn (op, e1, e2) =>
      Opn (op, eval (e1), eval (e2))
  }
```

In this version of `traverse` we evaluate eagerly, so that both sides of applications, the bound expressions and bodies of substitutions and the operands of primitives are evaluated. Forms that are not to be traversed do not need to be mentioned.

4.7 Congruences

Stratego provides a short-hand *congruence* notation for expressing traversal strategies of this kind. For example, if C is a node constructor with two arguments and s_1 and s_2 are strategies, then $C(s_1, s_2)$ is a congruence for C. It matches any C node, applies s_1 to the first component of the node, applies s_2 to the second component, and, if both s_1 and s_2 succeed, creates a new C node containing their results in the first and second components, respectively. If either s_1 or s_2 fail, then $C(s_1, s_2)$ fails.

`traverse` from the previous section can be written using Stratego congruences as follows.

```
App (s, s) + Let (id, id, s, s) + Opn (id, s, s)
```

The effect is to recursively evaluate those parts of the structure where `s` appears and leave untouched those parts where `id` appears.

Automatic support for congruences appears to beyond a pure embedding approach, since it requires knowledge of the abstract syntax. As a partial measure, Kiama helps developers of abstract syntaxes write their own congruences.[6] For example, a congruence for `App` can be written in Kiama as follows.

```
def App (s1 : => Strategy,
         s2 : => Strategy) : Strategy =
  rulefs {
    case _ : App => congruence (s1, s2)
  }
```

This definition overloads `App` to take strategy arguments. The pattern match restricts attention to `App` nodes and a Kiama library function `congruence` returns a strategy that implements the semantics of the congruence using `s1` and `s2`. `rulefs` is a variant of `rule` that takes a function returning a strategy instead of the usual function that returns a term. With this congruence definition, eager evaluation can be defined simply, without the supplementary `traverse` strategy.

```
val s : Strategy =
  attempt (App (s, s) + Let (id, id, s, s) +
           Opn (id, s, s)) <*
  attempt (lambda <* s)
```

4.8 Lazy Evaluation

Finally, we consider lazy evaluation where as much as possible is left un-reduced until a beta reduction is performed. Only the traversal strategy needs to change; the new traversal refrains from evaluating application arguments and let-bound expressions too early. (A full lazy evaluation method would also include sharing of computed values, which we omit here.) The change is restricted to the congruences, where `id` now appears in the positions for arguments to applications and bound expressions in `Let` constructs.

```
val s : Strategy =
  attempt (App (s, id) + Let (id, id, id, s) +
           Opn (id, s, s)) <*
  attempt (lambda <* s)
```

5 Conclusion

The examples in this paper are typical language processing problems: static analysis and evaluation by transformation. We have seen that these problems can be solved easily using embeddings of attribute grammars and strategic-based rewriting in a general purpose language. The attribute grammar embedding achieves a substantial proportion of the functionality of JastAdd and the rewriting embedding is very faithful to the Stratego language design. These embeddings can also be employed to solve other language processing tasks such as desugaring, interpretation, code generation, and optimisation.

[6] We plan to generate congruences from a description of the abstract syntax in a future version of Kiama.

The similarities between the embeddings of attribute grammars and rewriting show that these two paradigms are alike in ways that have not been appreciated to date. In each case, a simple functional interface, provided by `attr` and `rule`, suffices to hide the complexities of the representation of attributes and strategies. Apart from calling these functions, a Kiama programmer uses standard Scala constructs to define attribute equations and rewriting rules. Thus, Kiama's version of these paradigms is particularly lightweight compared to standalone generator-based systems such as JastAdd and Stratego. This lightweight nature makes it more accessible to mainstream developers who would otherwise not be exposed to these high-level processing paradigms. Moreover, since Kiama is pure Scala, it automatically gains advantage from existing Scala tools such as IDE support for editing and debugging, further simplifying the adoption process.

Kiama has some advanced capabilities that have not been presented here. The attributes used in this paper cannot have cyclic dependencies (i.e., depend on themselves). In some situations such cyclic dependencies are useful, particularly in analysis problems where a solution is found by computing until a fixed point is reached. See Sloane et al. [8] for an example that computes variable liveness information using a variant of `attr` designed to handle cyclic dependencies. Section 3.2 used a higher-order attribute which was a reference to an existing tree node; Kiama also allows higher-order attributes that refer to new nodes and supports *forwarding* [15] to redirect attribute evaluations automatically to higher-order attributes. Finally, Kiama includes *attribute decorators* [16] that can express patterns of attribute propagation.

Work on Kiama continues. Of particular interest is the interaction between the two paradigms, such as using the free variables attribute during rewriting. This kind of combination raises questions about the validity of attribute values after a rewriting step. We are exploring methods for removing the necessity for type casting of the generic structural properties such as `parent` due to a lack of knowledge about the tree structure. We are also investigating features such as collection attributes [17, 18] and better support for attribute modularity. Following Stratego, Kiama's strategies are currently largely untyped, except that Scala's type rules prevent ill-typed terms from being created. Typed strategies will come to Kiama soon. It would also be useful to have some way of specifying pattern matching on objects using concrete syntax [19, 20].

Acknowledgements

The author thanks the Software Engineering Research Group at the Technical University of Delft for hosting the study leave during which the Kiama project was initiated. That visit was supported by the Dutch NWO grant 040.11.001, *Combining Attribute Grammars and Term Rewriting for Programming Abstractions*. Lennart Kats and Eelco Visser co-developed the attribute grammar facilities described here and provided the author with instruction in the arts of Stratego. Charles Consel, Mark van den Brand, members of IFIP WG 2.11 and anonymous reviewers have also provided useful feedback on the Kiama project.

References

[1] Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Press (2008)
[2] Odersky, M.: Scala language specification, Version 2.7. Programming Methods Laboratory, EPFL, Switzerland (2009)
[3] Kastens, U., Sloane, A.M., Waite, W.M.: Generating Software from Specifications. Jones and Bartlett, Sudbury (2007)
[4] Hedin, G., Magnusson, E.: Jastadd: an aspect-oriented compiler construction system. Sci. Comput. Program. 47, 37–58 (2003)
[5] Visser, E.: Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In: Lengauer, C., et al. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 216–238. Springer, Heidelberg (2004)
[6] Lämmel, R., Visser, J.: A Strafunski Application Letter. In: Dahl, V. (ed.) PADL 2003. LNCS, vol. 2562, pp. 357–375. Springer, Heidelberg (2002)
[7] Lämmel, R., Peyton Jones, S.: Scrap your boilerplate: a practical design pattern for generic programming. ACM SIGPLAN Notices 38, 26–37 (2003); Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)
[8] Sloane, A.M., Kats, L.C.L., Visser, E.: A pure object-oriented embedding of attribute grammars. In: Vinju, J., Ekman, T. (eds.) Proceedings of the 9th Workshop on Language Descriptions, Tools and Applications (to appear in ENTCS) (2010)
[9] Deransart, P., Jourdan, M., Lorho, B.: Attribute Grammars: Definitions, Systems and Bibliography. LNCS, vol. 323. Springer, Heidelberg (1988)
[10] Paakki, J.: Attribute grammar paradigms—a high-level methodology in language implementation. ACM Comput. Surv. 27, 196–255 (1995)
[11] Jourdan, M.: An optimal-time recursive evaluator for attribute grammars. In: Paul, M., Robinet, B. (eds.) Programming 1984. LNCS, vol. 167, pp. 167–178. Springer, Heidelberg (1984)
[12] Reynolds, J.C.: Theories of Programming Languages. Camridge University Press, Camridge (1998)
[13] Hedin, G.: Reference Attributed Grammars. Informatica (Slovenia) 24, 301–317 (2000)
[14] Dolstra, E., Visser, E.: Building interpreters with rewriting strategies. In: Proceedings of the 2nd Workshop on Language Descriptions, Tools and Applications. Electronic Notes in Theoretical Computer Science, vol. 65, pp. 57–76 (2002)
[15] Van Wyk, E., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in attribute grammars for modular language design. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 128–142. Springer, Heidelberg (2002)
[16] Kats, L., Sloane, A.M., Visser, E.: Decorated attribute grammars: Attribute evaluation meets strategic programming. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 142–157. Springer, Heidelberg (2009)
[17] Boyland, J.T.: Descriptional Composition of Compiler Components. PhD thesis, University of California, Berkeley, Available as technical report UCB//CSD-96-916 (1996)
[18] Magnusson, E., Ekman, T., Hedin, G.: Extending attribute grammars with collection attributes - evaluation and applications. In: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation. IEEE Press, Los Alamitos (2007)

[19] van den Brand, M.G.J., van Deursen, A., Heering, J., Jong, H., Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF meta-environment: A component-based language development environment. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 365–370. Springer, Heidelberg (2001)

[20] Bravenboer, M., Visser, E.: Concrete syntax for objects. In: Schmidt, D.C. (ed.) Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2004), Vancouver, Canada, pp. 365–383. ACM Press, New York (2004)

Some Issues in the 'Archaeology' of Software Evolution

Michel Wermelinger and Yijun Yu

Computing Department & Centre for Research in Computing
The Open University, UK

Abstract. During a software project's lifetime, the software goes through many changes, as components are added, removed and modified to fix bugs and add new features. This paper is intended as a lightweight introduction to some of the issues arising from an 'archaeological' investigation of software evolution. We use our own work to look at some of the challenges faced, techniques used, findings obtained, and lessons learnt when measuring and visualising the historical changes that happen during the evolution of software.

1 Introduction

Wikipedia defines archaeology as "the science that studies human cultures through the recovery, documentation, analysis, and interpretation of material culture and environmental data." Software archaeology is similar: it studies the human activities during the lifetime of a software project through the artefacts it generates, such as code and design documents. As important are those intermediate items that often get lost after software delivery, in the form of emails, memos, tickets, drafts, comments or logs. Being the materials for archaeological study, these artefacts form important trails for others to reconstruct the history of development [7,15]. Their history is the fundamental memory for developers to maintain the software. They are also a valuable source for similar projects, showing good practice to be reused and pitfalls to be avoided.

All these artefacts are, ideally, kept in a readily interpretable and persistent form with the help of the 'digitalised memory' of the project life: version controlled repositories, archived emails, reported bug records, etc. However, just sieving through these data already imposes great challenges to software archaeologists, not to mention the additional difficulties in interpreting them. There are at least three types of transformations involved.

- 'Horizontal' or evolutionary transformations are conceptual units of work that lead from one version of the software to the next. These high-level transformations are often 'mediated' or triggered by other artefacts (e.g. bug reports) but the archaeological evidence of these transformations is often just fine-grained changes (e.g. lines of code added). As software processes are also programs [20], evolutionary transformations become amenable to analysis.

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 426–445, 2011.

- 'Vertical' or generative transformations are responsible for producing a lower-level artefact from a higher-level one. An example is the transformation from requirements to code via design documents. Code generation [8], meta-programming [10], and model-driven development [23], among others, are approaches that allow to specify and automate some of these transformations.
- Archaeological transformations involve the extraction and processing of data from available artefacts in order to turn such artefacts into models, measurements, visualizations and documentation that can help the archaeologist interpret the project's history.

Due to the different kinds of transformations involved, some common challenges faced when recovering and reconstructing the past are:

1. **Abstraction.** The sheer amount of data can easily overload a person [25]. The key is to use abstractions that serve the purpose at hand. One has to be careful though, as transforming the original data into its abstract form (e.g. a model or a metric) might lose subtle but important details.
2. **Lost artefacts.** Some artefacts are hardly documented, e.g. the original goals and motivations, assumptions, tacit knowledge, design rationale and principles [32]. However, they are what the archaeologists would like to infer from other related artefacts, at the risk of deriving wrong or biased information.
3. **Automation.** The archaeologist needs an assistant to perform the mundane work of data collection and transformation, otherwise they may not be able to understand the overall software system, because the history of artefacts is complex and expensive to find out manually. The assistant is ideally a robot that can perform these transformation tasks automatically. However, fully automated recovery is not always achievable depending on the source and target of the transformations [14].
4. **Evaluation.** Any analysis performed by one archaeologist should make sense to another archaeologist, but it may be more valuable if the evaluation is performed by the 'witnesses' (the original developers or users) or by alternative sources. Automated tools for measuring and visualising the software artefacts can be of a great help to make the study repeatable. The use of standard formats such as Rigi [29] to record the results would also help reduce the barrier for others to evaluate them.

To reconstruct a vivid history of software development, we aim to understand not only the deliverable artefacts such as the software itself, but also the tacit knowledge reflected by the communications and the coordination amongst stakeholders of the software. Recovering higher-level transformations from finer-grained changes would help the archaeologists reconstruct a model to sufficiently represent the evolution. Generative and archaeological transformations are not always in the form of round-trip to maintain the equivalence between the source and the target. Certain properties in the source can be preserved in the target, often with additional information at a lower level of abstraction. The reality is, however, that the information to recover transformations is not always available.

When transformations are applied in archaeology, one must be prepared for the loss of information, sometimes important one. As it is often difficult to obtain

the original sources when studying the target of these transformations, one must be careful not to put too much trust into the data sets. Only when reliable information is hard to obtain, one should rely solely on the data rather than the people. Open source projects [12] often make the life of archaeologists much easier as they make available not only the targets but also the sources and mediators of evolutionary and generative transformations, including the code, the email archives and the bug reports, etc.

In the next three sections we present a part of our own archaeological work on analysing architectural evolution and discuss in Section 5 some general issues and lessons arising from it. While our previous papers [26,27] focussed on the technical results (i.e. the outcome of the archaeological process), this one emphasizes the means to obtain them (i.e. the archaeological process itself) and the research path decisions taken. As such, Sections 1, 5 and part of 2 are new, and Sections 3.2 and 3.3 are the result of extensively rewriting, updating and expanding material that was fragmented across several papers [26,27,33], adding many more details about the data model and infrastructure used to assess the past history. Nevertheless, we also updated the results (Section 4), adding new data about the more recent releases of the chosen case study, and introducing a distinction between forced and unforced changes, which in turn led to several changes in the data visualization approach.

The paper, like the summer school that originated it, is aimed mainly at postgraduate students. By narrating our experience, and not just the end results, we aim to give those wishing to enter this research area a glimpse of the 'backstage' events of software archaeology. Interested readers are encouraged to afterwards consult more detailed treatments of this subject [19,17,24,7,15].

2 Motivation

Our research on architectural evolution started in a rather opportunistic way, when we came across the call for papers for the challenge track of the 5th Working Conference on Mining Software Repositories[1]. The challenge was to mine the Eclipse project, an open source integrated development environment (IDE) with respect to 1) bug analysis, 2) change analysis, 3) architecture and design, 4) process analysis or 5) team structure. The call for papers provided several CVS repositories with subsets of the Eclipse project, but authors could choose any other data source.

Given our past interest in software evolution and software architecture, and knowing that Eclipse had a strong IBM lead, we decided to attempt an analysis of the architectural change process, thereby addressing topics 2), 3) and 4) of the five proposed[2]. To be more precise, the research questions that we had in mind were:

[1] `http://msr.uwaterloo.ca/msr2008/challenge/`

[2] We later also looked briefly into the team structure of Eclipse and how it changed over time [28], but will not go further into it for this paper.

1. Is there any systematic architectural change process, or is the architecture being continually modified in every release?
2. Does the architectural evolution follow any of Lehman's software evolution laws, like continuous growth and increased complexity?
3. Is there any evidence of restructuring work aimed at reducing growth and complexity?
4. Is there any stable (i.e. unchanged) architectural core around which the system grew?

Once we saw the Eclipse project contained data to enable the archaeological investigation of those questions [26], the next step was to widen the scope of the research questions, looking at whether Eclipse's architecture was following design guidelines that have been proposed to ease changes, like absence of cyclic dependencies, low coupling and Martin's stable dependency principle [18]. The detailed motivation, research questions and results of those investigations were presented in [27]. Here, we only revisit one of the questions:

5. Does cohesion increase and coupling decrease over time?

The main point to keep in mind is that, while the research was *triggered* by a given case study, it was *led* by general research questions about architectural process and design principles. The overarching motivation was to *invalidate* such principles, in the spirit of falsifiability of scientific hypotheses [21]: if a highly successful and continuously evolving infrastructure project like Eclipse, on which many third-party components and applications have been built, does *not* follow commonly recommended guidelines, the usefulness (or at least the importance) of such guidelines could be questioned.

3 Data Collection

After having explained our motivation and particular architectural research angle, we can look more closely at the case study, which data was extracted, and how.

3.1 The Case Study

The case study consists of multiple *builds*, i.e. snapshots, of the Eclipse Software Development Kit (SDK) source code. Each build is implemented by a set of *plugins*, Eclipse's components. Each plugin may *depend* for its compilation on Java classes that belong to other plugins. For example, the implementation of plugin platform (we omit the default org.eclipse prefix) in 3.3.1.1 depends on eight other plugins, including core.runtime and ui. Each plugin *provides* zero or more *extension points*. These can be *required* at run-time by other plugins in order to extend the functionality of Eclipse. A typical example are the extension points provided by the ui plugin: they allow other plugins to add at run-time new GUI elements (menu bars, buttons, etc.). It is also possible for a plugin to use the extension

Fig. 1. Chronological and logical sequences of some of the analysed builds

points provided by itself. Again, the ui plugin is an example thereof: it uses its own extension points to add the default menus and buttons to Eclipse's GUI.

In the remaining of the paper, we say that plugin X *statically depends* on plugin Y if the compilation of X requires Y, and we say that X *dynamically depends* on Y if X uses at run-time an extension point that Y provides. Note that the dynamic dependencies are at the architectural level; they do not capture run-time calls between objects.

For our purposes, the architectural evolution of Eclipse corresponds to the creation and deletion of plugins and their dependencies over several builds. There are various types of builds in the Eclipse project. We analysed *major and minor releases* (e.g. 2.0 or 2.1) and the *service releases* that follow them (e.g. 2.0.1). In parallel to the maintenance of the current release, the preparation of the next one starts. The preparation consists of some *milestones*, followed by some *release candidates*. For example, release 3.1 was followed by milestone 1 of release 3.2 (named 3.2M1), further five other milestones, and seven release candidates (3.2RC1, 3.2RC2, etc.), culminating in minor release 3.2.

Figure 1 shows part of the builds we analysed, and their chronological and logical order. The logical order is indicated by solid arrows: each release may have multiple logical successors. The chronological order is represented by positioning the nodes from left to right: each release has a single chronological successor. The dotted arrows indicate that some builds, in which the chronological and logical orders coincide, were omitted due to page width constraints.

For our purposes, it makes more sense to order the builds by their numbers rather than by their dates, i.e. to follow a logical rather than a chronological order. The latter is useful when analysing the amount of changes per fixed time frame, which is for example necessary if one wishes to compare the evolution of different systems [12]. In our case, due to research question 1 (Section 2), we wish to check whether architectural changes are associated to particular builds. Hence, we compare changes between builds in logical order. For example, instead of analysing the chronological sequence 3.1, 3.2M1, 3.2M2, 3.1.1, 3.2M3, 3.2M4, 3.1.2 (see Figure 1), we either follow the main sequence 3.1, 3.1.1, 3.1.2 or the milestone sequence 3.1, 3.2M1, 3.2M2, ..., 3.2. For this paper we analysed two build sequences: the 26 major, minor and service releases from 1.0 to 3.5.1 over a period of almost 8 years (from November 2001 to September 2009), and the 27 milestones and release candidates between 3.1, 3.2, and 3.3 over a period of 2 years (from June 2005 to June 2007).

3.2 The Data Model

To perform our analyses in a systematic way and to be able to reapply them to other case studies, we define a very simple structural model and associated

metrics. We were inspired by an existing axiomatic metrics framework [5], in which a generic structural model serves to impose constraints to characterize different kinds of metrics (size metrics, cohesion metrics, etc.). Our structural model is simpler and our metrics largely follow the constraints proposed in [5].

We represent a *module* (to use a relatively neutral term) by a directed graph, where nodes represent elements and arcs represent a binary relation between elements. Each element is classified as being either *internal* or *external* to the module. Likewise, internal relationships *IR* are those between internal elements *IE*, while external relationships *ER* are those between an internal and an external element *EE*. In this way, the description of a module also includes the connections to its context. Formally, a module is a graph $G = (IE \cup EE, IR \cup ER)$, such that $IE \cap EE = \emptyset$, $IR \subseteq IE \times IE$, and $G' = (IE \cup EE, ER)$ is a bipartite graph.

We define the following metrics on modules.

- The *size* of a module is the number of internal elements: $\text{size}(G) = |IE|$.
- The *complexity* is the number of internal relationships: $\text{complexity}(G) = |IR|$. Since it is impossible for a single metric to fully capture complexity, our aim was to define it as simply and as generally as possible.
- The *cohesion* could be defined as the ratio between the complexity and the square of the size. The reason for this definition is for cohesion to be normalised and to reach its maximal value for complete graphs. Given that we should not expect a well designed architecture to evolve towards a complete graph, we define the metric instead as to be a simple relationships to elements ratio: $\text{cohesion}(G) = \text{complexity}(G)/\text{size}(G)$.
- The *coupling* of a module is the number of (incoming and outgoing) external dependencies: $\text{coupling}(G) = |ER|$.

The graph-based model is generic enough for modules, elements and relationships to represent almost anything. For example, modules and elements can represent Java packages and classes, respectively, with arcs representing the inheritance relation. A module may also correspond to a class, with elements representing methods and arcs representing the call relation.

For our purposes, we wish to apply the model to Eclipse and other plugin-based architectures. Therefore, we take a module to be the whole architecture of a sub-system (the Eclipse SDK in this case study) and an element to be a plugin, while relationships may denote the static or dynamic dependencies. Because of the latter, we also need to include in the model the extension points provided and required by each plugin.

We use a relational representation instead of a graph-based one, for practical reasons. Operationally, the first step consists of defining the following relations from the repository's data:

- $IP(p)$ or $EP(p)$ holds if p is an internal or external plugin
- $Prov(p, e)$ or $Req(p, e)$ holds if plugin p provides or requires extension point e
- $SD(p, p')$ holds if plugin p statically depends on plugin p'

From these, the following relations can be computed:

- internal static dependencies $ISD(p,p') \equiv SD(p,p') \wedge IP(p) \wedge IP(p')$
- external static dependencies $ESD(p,p') \equiv SD(p,p') \wedge \neg ISD(p,p')$
- dynamic dependencies $DD(p,p') \equiv \exists e : Prov(p',e) \wedge Req(p,e)$
- internal dynamic dependencies $IDD(p,p') \equiv DD(p,p') \wedge IP(p) \wedge IP(p')$
- external dynamic dependencies $EDD(p,p') \equiv DD(p,p') \wedge \neg IDD(p,p')$
- internal dependencies $ID(p,p') \equiv ISD(p,p') \vee IDD(p,p')$
- external dependencies $ED(p,p') \equiv ESD(p,p') \vee EDD(p,p')$

Given the above relations, computing the metrics is just a matter of computing the cardinality (i.e. the number of tuples) in the appropriate relation. For example the size is $|IP|$ and the complexity is $|ISD|$ or $|IDD|$ or $|ID|$, depending on which dependencies we take as the arcs. Note that in general $|ID| \leq |ISD| + |IDD|$.

The relational model further allows to compute missing (i.e. required but not provided) and unused (i.e. provided but not required) plugins and extension points. For example, given all plugins $P(p) \equiv IP(p) \vee EP(p)$ and all dependencies $D(p,p') \equiv SD(p,p') \vee DD(p,p')$ we have

- missing plugins $MP(p) \equiv \exists p' : D(p',p) \wedge \neg P(p)$
- unused extension points $UEP(e) \equiv \exists p : Prov(p,e) \wedge \neg \exists p' : Req(p',e)$

Missing artefacts indicate potential compile-time or run-time errors, or an ill-defined module boundary, or some problem with the data mining process. Unused artefacts tell us how open and extensible the module is. Too many unused elements such as unused extension points provided by the internal plugins, might be an indication of premature generality. A completely self-contained and closed module would have no missing nor unused elements.

To allow a historical analysis, the model has to be enriched with the notion of a snapshot, which is a module at some point in time. For our case study, a snapshot is one of the Eclipse builds mentioned in Section 3.1. All the above relations must have an additional argument stating the snapshot in which they hold. For example, $P(p,s)$ holds if plugin p exists at snapshot s and $SD(p,p',s)$ holds if p statically depends on p' in snapshot s. To allow flexibility in the choice of the snapshot sequences to analyse, we allow the researcher to define the relation $Next(s,s')$, which states that snapshot s' comes immediately after snapshot s. The relation is considered ill-defined if a snapshot succeeds itself, has more than one successor, or if more than one snapshot has no predecessor. The unique snapshot without predecessor is considered the first release of the sequence: $First(s') \equiv \exists s'' : Next(s',s'') \wedge \neg \exists s : Next(s,s')$.

Once a sequence is defined, it is possible to compute how each module snapshot has been obtained from the previous one. In particular, we compute:

- added plugins $AP(p,s') \equiv P(p,s') \wedge Next(s,s') \wedge \neg P(p,s)$
- kept plugins $KP(p,s') \equiv P(p,s') \wedge First(s) \wedge P(p,s)$
- deleted plugins $DP(p,s') \equiv Next(s,s') \wedge P(p,s) \wedge \neg P(p,s')$
- previous plugins $PP(p,s') \equiv P(p,s') \wedge \neg AP(p,s') \wedge \neg KP(p,s')$

and similarly for static and dynamic dependencies. This of course assumes that elements and relations maintain a unique name throughout the module's history, which means that a renaming will be counted as a simultaneous deletion and addition. The aim of computing the kept (i.e. unchanged) elements and relationships of the module is to address Question 4 in Section 2.

3.3 The Tool Infrastructure

We developed a suite of small tools that first extract the data, then compute the metrics, and finally visualise the results. However, we took care to make the suite relatively independent of our particular needs, in order to be useful in a variety of contexts. Therefore, instead of developing a standalone application or an extension for a particular IDE, we have put together a simple pipeline architecture of scripts that manipulate text files. This makes it easier to interface with other tools and to replace part of the pipeline, e.g. for a different case study.

A partial architecture of our tool suite[3] is shown in Figure 2 as a set of processes that convert input data files on the left into the output data files on the right. Among the processes, *fact extractors* obtain factual relations from artefacts of a single release of the software system and store the relations in Rigi Standard Format (RSF) files. RSF is a simple and widely used text format in which each line represents a tuple, with the relation name being followed by each tuple element, separated by spaces [29]. We next used the relational calculator Crocopat [3] to implement a *fact merger* that combines facts about selected individual snapshots into a single fact base by adding the snapshot id to every relation tuple. *Metric calculators* compute from the fact base a number of metrics, such as size and complexity. The *reporters* present the metrics and the architecture in a number of ways, including various visualisations. In the remaining of this section we detail parts of the mining process.

For each Eclipse plugin there is an XML file, called `plugin.xml`, that lists the extension points provided and used by that plugin, and the other plugins it depends on for compilation. Since release 3.0, the static dependency is in another file, `MANIFEST.MF`, which is not in XML format. These metadata files are hence a straightforward source of dependency information between plugins, saving us from having to delve into their source code. We fully agree with Alex Wolf's argument in his WICSA'09 keynote, that configuration files are an underexplored source of architectural information, which has so far been mainly extracted, in a potentially not very reliable way, from source code.

We first considered extracting the metadata files for each build directly from the CVS repository, for example by checking out all files with tag `R_3_1` (CVS tags cannot include periods) in order to obtain the information about release 3.1. However, after a while we found out that there is no direct correspondence between CVS tags and builds. In other words, comparing the set of metadata files obtained from the CVS repository with the set of those included in the actual builds, we found that often the two sets didn't coincide. We also tried to

[3] The complete suite also includes the mining of Bugzilla repositories [28].

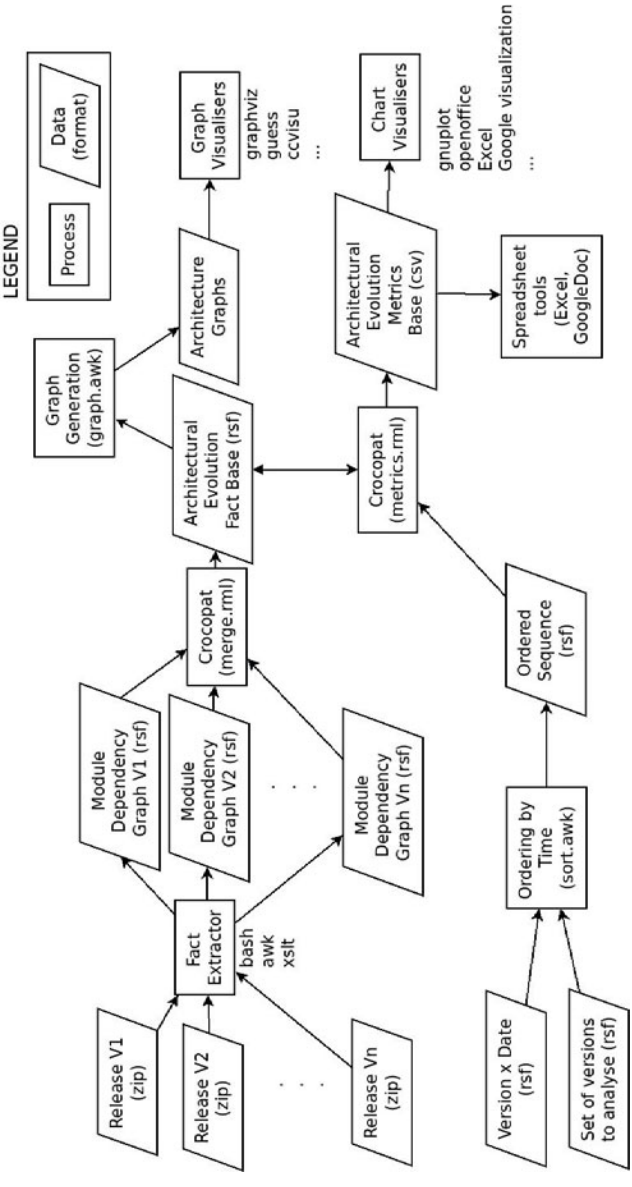


Fig. 2. Overview of our toolset

check out the files according to the known date of the build, but again there was a mismatch. We realized the Eclipse project uses for each build a complicated file that indicates which source files are included.

The input to our analysis is therefore not a CVS repository, but a set of compilable source code archives, one per build we wish to analyse. How each

source code archive was obtained is not of concern to our tool, making it independent of any configuration management system. In our case, for each of the builds we analysed, we downloaded the source code of the whole SDK from http://archive.eclipse.org or its mirrors. In previous work [26,27] we only analysed builds up to 3.3.1.1. When starting to download more recent ones for this paper, we were dismayed to find out that the Eclipse project no longer keeps older milestones and release candidates in their archive, in order to save storage and bandwidth. We therefore were only able to add releases (6 of them) for this paper. Fortunately, we kept a copy of the previously downloaded milestones and release candidates, enabling us to do further mining on them.

The repository is first processed by some shell, AWK and XSLT scripts that extract the information about the existing architectural elements from the `plugin.xml` files (and `MANIFEST.MF` files, depending on the build). The result of this processing is a RSF file with the basic relations (IP, EP, $Prov$, Req and SD) presented in Section 3.2. Whereas in previous work we defined as an internal plugin any component for which a plugin.xml file existed, in this paper we only take the subset of those where the name starts with org.eclipse but does not end in source. A source plugin wraps the source code of some other plugin, so that the code can be accessed for help and debugging purposes in the Eclipse IDE, by providing extensions to the pde.core plugin. Given that source plugins don't add functionality, we decided to ignore them for this study. Moreover, since in recent releases many plugins also have their source counterpart, this would greatly inflate the metrics, in particular the size metric.

Once we have the basic relations for each snapshot, we use Crocopat first to merge all RSF files into a single one (top left of Figure 2) as mentioned before, and second to compute any derived relations and metrics (Section 3.2 and centre of Figure 2), given the snapshot sequence. For example, from the $Prov$ and Req relations between plugins and extension points, a Crocopat script computes the dynamic dependency relation among plugins. Crocopat is also used to compute transitive closures over dependencies, in order to detect dependency cycles. The Crocopat script also computes added, deleted and kept plugins and dependencies, distinguishing between unforced and forced additions and deletions. We will explain those concepts in Section 4.

Finally, for the 'front end' of the chain, we use Crocopat and AWK to automatically translate the relevant relations in the RSF files (e.g. SD) into files for input to graphviz[4], GUESS [1] and CCVisu [2]. This allows to display or animate the architectural structure in various ways. As for showing the evolution of metrics along build sequences, we simply use bar and line charts. In previous work we used Crocopat to generate spreadsheets in OpenOffice's XML format and used OpenOffice or Excel to create the charts. For this paper we took another path: Crocopat generates comma separated value files (one for each sequence) which we upload to Google Spreadsheets. We then wrote Javascript code that calls

[4] `http://www.graphviz.org/`

the Google Visualization API[5] in order to get the data from the spreadsheets, generate charts and embed them into a web page.

Using Google tools has several advantages over the previous approach. First, the data is made public to other researchers and in various formats (HTML, OpenOffice, Excel) without any additional effort on our part. Second, the bar and line charts are large and interactive, allowing the reader to click on the data points to see the exact values, instead of just perceiving generic trends from a small, static, and grey chart in a paper. Third, the Google Visualization API includes an expressive data query language that allows some calculations to be performed on the fly, like computing the ratio of the values in two columns. This means that some additional metrics can be presented without having to change the Crocopat script, run it again and upload the new spreadsheet.

Overall, our tool infrastructure has been designed and developed over time with the aim of being flexible, light-weight and interoperable. Flexibility and interoperability are achieved by an open and easy to modify pipe-and-filter architecture in which the pipes are text files in standard formats (XML, RSF) and the filters are scripts executed by widely used, freely available, and generic data processing and visualization tools (AWK, XSLT, Crocopat, graphviz, etc.). Due to this, it should not be too difficult to integrate our scripts within existing tool chains, like FETCH [4], and to modify the 'back-end' to handle other systems besides Eclipse.

The approach is light-weight because it is independent of any particular configuration management tool like CVS or Subversion, because it just relies on metadata files and not on static code analysis, and because the relations are kept in text files. Given the small size of the database (87,397 tuples for the 53 Eclipse builds analysed), our approach remains very efficient.

4 The Results

After presenting the data model and how the data is mined and processed, we are in a position to show the results. The charts presented in this section (and others) can be interacted with at a web page[6] that also links to the spreadsheets with all measurements.

To show the evolution of the metrics over the two snapshot sequences, we use mostly stacked bar charts, with each bar segment showing a particular subset of the total number of items (plugins or dependencies). The segments are stacked, from bottom to top, as follows: unforced deletions, forced deletions, kept items (i.e. since the first snapshot in the sequence), previous items, forced additions, unforced additions. In general, a change is considered unforced if it is by choice, and forced if it is due to another change, e.g the unforced deletion of a plugin forces the deletion of all its extension points and dependencies.

[5] `http://code.google.com/apis/visualization`

[6] `http://michel.wermelinger.ws/chezmichel/2009/10/the-architectural-evolution-of-eclipse`

We use the same colour for unforced additions and deletions, and the same colour for forced deletions and additions. Since deletions are represented by negative numbers and additions by positive ones, there is no possible confusion. We also use a darker colour to distinguish kept from previous items. On the PDF version of this paper you can see we use warmer colours (red and orange) for changed and cooler colours (blue tones) for unchanged items. The aim of these choices was to have a reduced colour palette that translated well to grey scale values in the printed version, while using position and hue to quickly draw the reader's attention to the unforced changes at the extremities of each bar.

4.1 Size

Figure 3 shows the evolution of Eclipse's size, along the two snapshot sequences. Note that the number of kept plugins is with regard to the first release in the sequence, i.e. 1.0 or 3.1. We consider all plugin additions and deletions as unforced, because they are architectural choices.

We can observe that, over all releases, the size of the architecture increases more than sevenfold, from 35 to 271. The evolution follows a segmented growth pattern, in which different segments have different growth rates. In particular, the rate is zero during service and positive during major and minor releases. A look at the interim builds reveals that most of those changes occur in milestones, although some also occur in the later release candidates.

Segmented growth patterns have been observed for other open source systems, as surveyed in [12]. Those studies also observed superlinear growth, i.e. growth with increasing rates, which is not the case here. Our hypothesis is that while those studies focused on source code, we focus on the architecture, which, to remain useful and understandable to stakeholders, has to be kept within a reasonable size. In fact, the evolution of the size follows a pattern observed for other systems [30]: long *equilibrium periods*, in which changes can be accommodated within the existing architecture, alternate with relatively short *punctuation periods*, in which changes require architectural revisions.

4.2 Complexity

Figure 4 plots the changes to overall complexity, i.e. to relation ID (Section 3.2). The web page indicated earlier provides additional charts for static and dynamic internal dependencies and for milestones and release candidates. A forced addition or deletion of a dependency is associated to the creation or removal of at least one of the involved plugins, i.e. the addition (resp. deletion) of a dependency between two plugins is called unforced if both plugins already existed (resp. still remain).

Again, dependencies change mostly during milestones and remain the same during service releases, except for a few deletions in 3.3.1. However, contrary to continuous increase of size, there has been a decrease of complexity in release 3.1, i.e. there was some effort to counteract the system's growth.

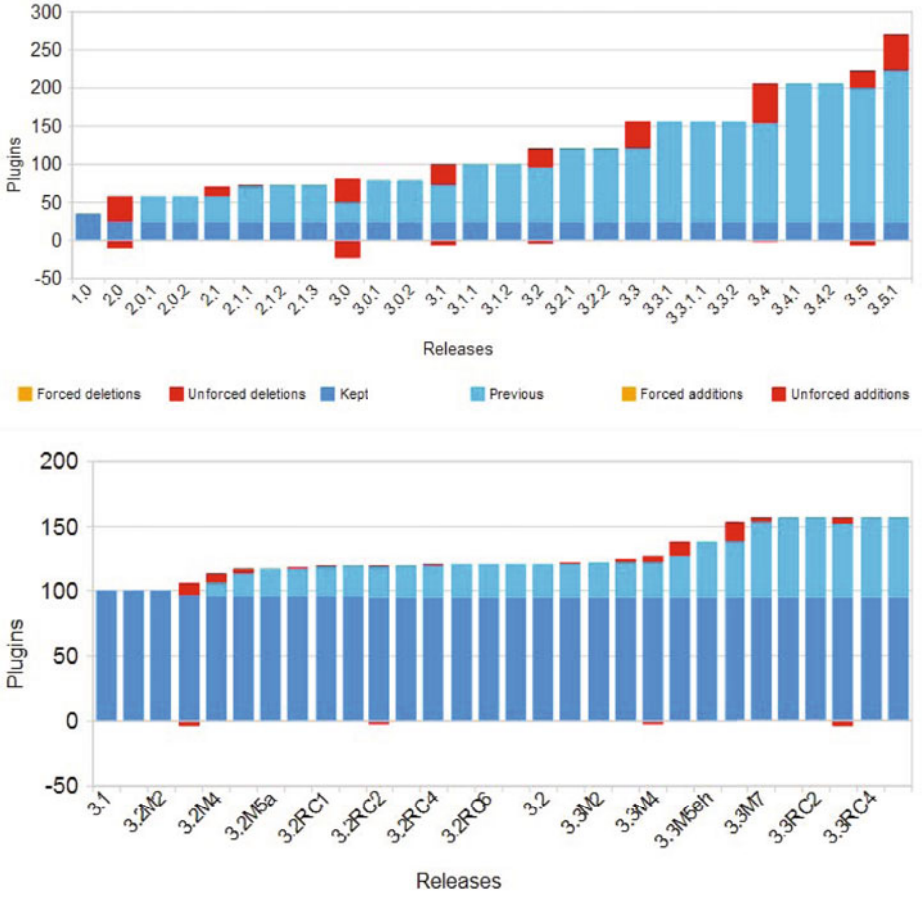


Fig. 3. Evolution of the size

Moreover, the chart shows that most additions are forced, i.e. new dependencies are due to new plugins, while most deletions are unforced, i.e. due to changes in the plugins' implementations in order to reduce dependencies.

The new releases analysed for this paper continue to keep the same plugins and dependencies since release 1.0, as seen by the continuous dark blue segments in Figures 3 and 4. The architectural core is hence the same as presented in [27].

4.3 Cohesion

The previous charts show that size and complexity grow 'in sync', following the same punctuation and equilibrium pattern. There are however two exceptions: release 3.0 substantially increased the complexity while only slightly increasing the size, and release 3.1 decreased the complexity while increasing the size.

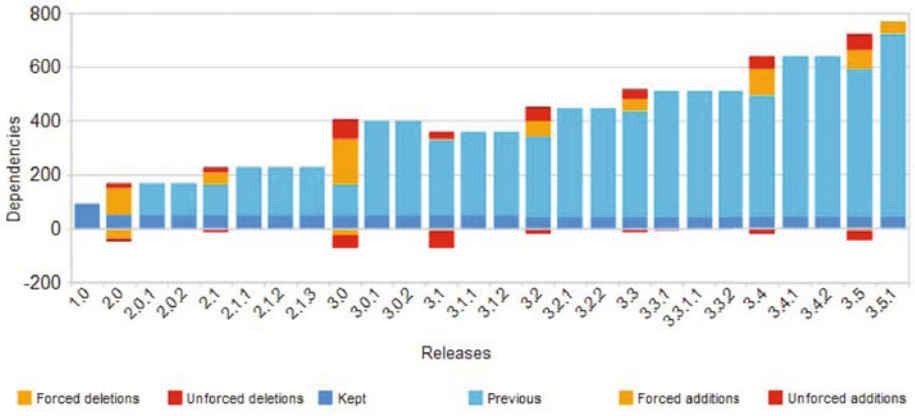


Fig. 4. Evolution of the overall complexity

Hence, computing the cohesion, we note it is remarkably almost constant (Figure 5) except for the increase at 3.0, which was kept until 3.1 because service releases didn't change the architecture. After 8 years, the cohesion levels of release 3.5.1 (1.40 internal dynamic dependencies and 2.17 static ones per plugin) are very similar to those of the much smaller release 1.0. The chart also shows that there are many more static dependencies than dynamic ones, as can be checked with the additional complexity charts on the web site mentioned earlier.

Interestingly, when we showed the previous version of this chart [27] to Eclipse developers at IBM Zurich, we were told there was no explicit aim to keep the cohesion constant. Nevertheless, we conjecture this might be an indirect consequence of possibly wishing to keep the various Eclipse SDK sub-systems (the Plugin Development Environment, the Java Development Toolkit, etc.) loosely cohesive to facilitate the configuration of the IDE to individual needs.

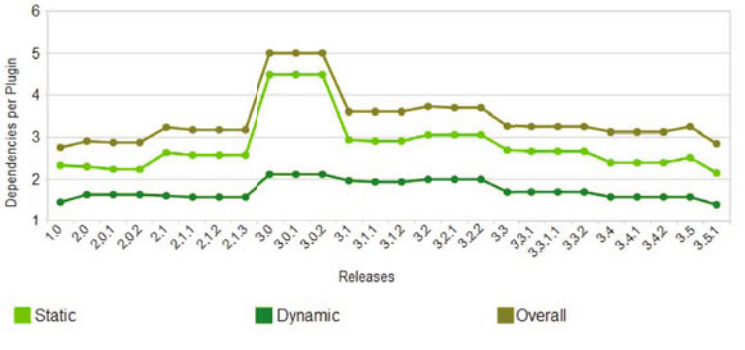


Fig. 5. Evolution of the cohesion

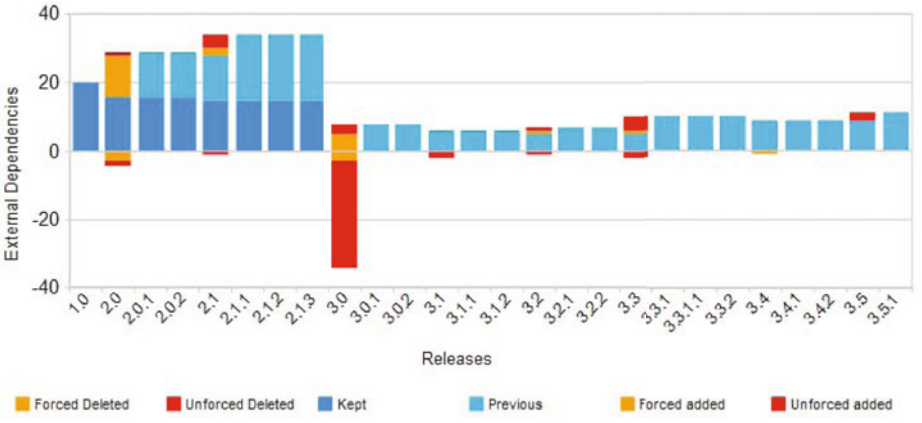


Fig. 6. Evolution of the coupling

4.4 Coupling

The evolution of coupling also follows a segmented growth pattern, but with a substantial decrease in release 3.0, which replaced all external dependencies (Figure 6). Release 3.1 further reduced the dependency on external plugins, although it grew again in later releases.

We looked into the actual dependencies and plugins involved, and realized that plugins that depended on external plugins in 2.1.3, depend in 3.0 on new internal plugins which in turn depend on the external plugins. In other words, release 3.0 introduced internal 'proxy' plugins for the external plugins, and this reduced coupling between Eclipse and third-party components. Additionally, one of the external plugins used by release 2.1.3, org.apache.xerces, was removed. Figure 6 sums up all these modifications as unforced changes (the rewiring) and forced changes (due to the removed plugin and new proxies). Overall, the chart shows most changes to the coupling are unforced, i.e. by choice rather than due to the addition or removal of plugins.

4.5 Summary

We can now return to the initial questions (Section 2) and summarize what the archaeological investigation has shown.

1. The development of Eclipse follows a systematic process in which the architecture is mainly changed during the milestones of the next major or minor release. Some release candidates may still introduce some small changes, but the architecture is frozen for the last few builds before the release. Service releases almost never introduce any architectural changes.
2. Overall, the Eclipse architecture is always growing and as such follows Lehman's 6th law of evolution. Due to the systematic change process, such growth follows a known segmented pattern of alternating long equilibrium

and shorter punctuation periods, the latter mostly during milestones. Complexity (as measured by the dependencies among plugins) also increases, as Lehman's 2nd law postulates, and does so following the same segmented growth pattern as size.
3. There has been some effort to reduce the system's growth, but overall deletions are far fewer than additions, possibly to avoid breaking the many existing third-party Eclipse plugins. The major reduction efforts have been in releases 3.0 (small size growth, reduced coupling) and 3.1 (reduced complexity and coupling).
4. The new releases analysed for this paper continue to use the layered architectural core we presented before [27].
5. The Eclipse architecture is kept loosely cohesive during its evolution, contrary to our initial expectations. However, Eclipse developers follow the usual advice of minimising coupling: the number of external static dependencies is very small compared to the number of internal ones and there have been explicit efforts (i.e. unforced changes) to reduce coupling.

To sum up, we were not able to find any empirical evidence to falsify the investigated design guidelines and evolution laws, with the possible exception of increased cohesion. From the above observations (systematic process, segmented growth, punctual but extensive restructurings, and avoidance of deletions), we feel that Eclipse can be used as a pedagogical case study of best practice to achieve sustainable architectural evolution of software frameworks.

5 Discussion

Reflecting on our work, we can pass on several lessons and issues to be aware of when embarking on an archaeological investigation of software evolution.

For brevity and clarity, most research papers only present the results, i.e. the 'after the fact' picture of the research process, in which all pieces of the puzzle fit into a perfectly logical conceptual building. The dead ends and twists and turns of the path that led to the results remain often unreported. In reality, the process is not as linear as the papers, written on hindsight, seem to imply. In particular, software archaeology is iterative and incremental, with a constant interplay between research questions, which provide the overall guidance, and the available data, which constrains what can be done. Both parts mutually influence each other and together shape the overall data mining and analysis. For example, while we presented the research questions (Section 2), the data model (Section 3.2) and its extraction from the Eclipse repository (Section 3.3) in a sequential fashion, each phase apparently determining the next one, in reality a preliminary analysis of the repository was needed to assess what data could be extracted, i.e. what kinds of builds and architectural information was available, which in turn helped shape the research questions, the abstracted data model and the tool set. Software archaeologists must therefore be prepared to 'follow' the data, especially if faced with lost artefacts, as mentioned in the introduction. Like in real archaeology, software-related artefacts may be lost because they

were not recorded in a persistent way, or because they were later 'destroyed' by accident or on purpose. Keeping your own copy of datasets might be a good idea, as we found out (Section 3.3).

However, the research cannot be completely data-driven. Software repositories are simply too rich and big for an ad-hoc exploration to guarantee interesting results with little effort. Research questions are hence fundamental to frame and guide an efficient mining process. Moreover, questions must be explicit and relevant in order to avoid the dreaded 'so what?' question by critics. Relevancy can be pedagogical, practical or theoretical, often being a mix of the three, as in our case: while the main aim was theoretical, seeking empirical evidence of design guidelines and evolution laws, the results can be used for teaching purposes, using Eclipse as a good practice exemplar of architectural evolution.

The mining infrastructure should also be a reusable asset. Tool development takes considerable effort; return on investment is obtained by using the tools over several research iterations. Moreover, one should strive to build upon third-party infrastructure. Examples of reusable tools that build upon other existing tools are MoDisco[7], an Eclipse plugin, and the batch-oriented tool chain FETCH [4]. Both approaches have advantages. Tools within IDEs become part of the developers' workflow: the archaeological process is tightly integrated with the development process, each one feeding into the other. IDE-independent tools like FETCH can be more general and flexible, because wiring together existing generic data processing and visualization tools allows adaptation to a variety of research scenarios and data sources. On the other hand, tools like MoDisco aim to achieve such flexibility by providing a generic model transformation infrastructure that is able to generate metrics, visualizations and documents from models, allowing users to tailor the models and transformations to their particular needs. While our approach is also driven by a model (of the system's structure), it is ad-hoc in the sense that the model and metrics, albeit generic, are fixed, whereas a truly model driven approach like MoDisco is much more customizable, systematic, expressive and reusable. However, such characteristics come at a price: model-driven approaches require heavy-weight infrastructure and considerable investment from the user to learn and customize it, even for simple models and measurements like those in this paper.

Once the data has been mined and processed, it has to be presented. Simple quantitative displays (e.g. line diagrams) are a good indication of the change rate, but visualising the actual transformations (e.g. the before and after architecture) still poses a challenge, even if using animations. As the charts in Section 4 indicate, even at the highest level of design abstraction, any realistic system comprises hundreds of artefacts. Presenting them in an understandable way on a big screen is challenging, let alone on paper. We have experimented with graphviz and GUESS, but results were unsatisfactory. Only graphs with relatively few nodes and arcs, like the architectural core, can be easily depicted.

Contrary to Physics and other subjects, there is not yet a culture in Computing that leads authors to fully publicise the data on which their conclusions

[7] `http://www.eclipse.org/gmt/modisco`

are based, so that other researchers can build on it and independently verify it. Publishers do not yet provide the means for such data to be stored and accessed as easily as the papers that report on the data. Fortunately, due to the Web 2.0 it is becoming easier for authors to publish their data and visualizations, and we described one way to do so in Section 3.3.

Tracking changes in artefacts is a long-standing research strand. As mentioned in the introduction of this paper, one of the issues is abstraction, in particular how to abstract fine-grained changes into meaningful transformations. One possible heuristic is to attempt to minimize the number of transformations that encompass all observed changes. Two approaches that follow such a strategy for source code changes are [6,13]. Those proposals appeal to the language engineering community [16,9] where the primary artefacts are text-based.

On the other hand, when the artefacts are structured as models, one may leverage more semantic information (e.g. from UML model elements and their relationships) to detect structural changes [31,22]. Such approaches appeal to the model-driven engineering community because the basic changes detected can suggest more complex adaptive framework changes [11] at the modeling level.

Whereas text or model comparison reconstructs the actual changes, measuring changes is a good way to spot overall trends. One particularly relevant trend for evolution is zero changes, i.e. what does *not* change. In our work, it corresponds to the architectural core, an important design feature.

However, metrics don't tell the whole story: they don't capture all the 'what' and 'how' of evolution and certainly not the 'why'. Hence, measurements should be complemented by an inspection of the actual artefacts and, if possible, by other information sources, e.g. bug reports or the system's developers. For example, metrics and the distinction between forced and unforced changes can tell that Eclipse was restructured in release 3.0, but only looking at plugins can one understand it was in part due to the adoption of the OSGi run-time infrastructure. Also, without asking the developers one might assume that the constant cohesion is a deliberate design aim.

In spite of all sources of information one can consult, researchers and their audience must accept that in software archaeology there will always be some space for subjective interpretation, first because, contrary to apples falling on scientists' heads, software projects don't follow any natural laws, and second because threats to the validity of the conclusions can hardly be completely eliminated. There might be errors in the mining infrastructure, the statistical method employed might be inappropriate for the data at hand, etc. As in the natural sciences, any abstraction/model can only provide a partial view on the studied subject, and software development is a complex socio-technical endeavour with many potential confounding factors. In our case, the simple size, complexity and cohesion metrics only provide a very partial view of software architecture.

Researchers often strive to justify they adequately handled the threats to validity, but it is probably sometimes better to just point to them as opportunities for further improvement. After all, research is a community practice, not an individual pursuit.

6 Conclusions

This paper is a twofold tutorial. On the one hand, in a similar spirit to the case studies presented in business management literature, the research provides empirical evidence for using Eclipse as a tutorial case study on sustainable good practice for architectural evolution. On the other hand, the research serves as a tutorial-by-example on some of the issues faced when doing archaeological investigations into software evolution. For that, we provide more details on the research process than in our previous papers, make the measurements publicly available, and reflect on our experience.

Software archaeology, not just for evolution, is blooming due to the increased availability of rich software project repositories. We hope this paper helps the next generation of researchers in this exciting area.

References

1. Adar, E.: GUESS: a language and interface for graph exploration. In: Proc. SIGCHI Conf. on Human Factors in Computing Systems, pp. 791–800. ACM, New York (2006)
2. Beyer, D.: CCVisu: automatic visual software decomposition. In: Proc. of Int'l Conf. on Software Engineering, companion volume, pp. 967–968. ACM, New York (2008)
3. Beyer, D., Noack, A., Lewerentz, C.: Efficient relational calculation for software analysis. IEEE Trans. Software Eng. 31(2), 137–149 (2005)
4. Bois, B.D., Rompaey, B.V., Meijfroidt, K., Suijs, E.: Supporting reengineering scenarios with FETCH: an experience report. Electronic Communications of the EASST 8 (2008)
5. Briand, L.C., Morasca, S., Basili, V.R.: Property-based software engineering measurement. IEEE Trans. Software Eng. 22(1), 68–86 (1996)
6. Canfora, G., Cerulo, L., Di Penta, M.: Tracking your changes: A language-independent approach. IEEE Softw. 26(1), 50–57 (2009)
7. Canfora, G., Penta, M.D.: New frontiers of reverse engineering. In: Future of Software Engineering, pp. 326–341. IEEE, Los Alamitos (2007)
8. Cordy, J.R.: The TXL source transformation language. Sci. Comput. Program. 61(3), 190–210 (2006)
9. Cordy, J.R.: The txl source transformation language. Sci. Comput. Program. 61(3), 190–210 (2006)
10. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional, Reading (June 2000)
11. Dagenais, B., Robillard, M.P.: Recommending adaptive changes for framework evolution. In: Proc. 30th Int'l Conf. on Software Engineering, pp. 481–490. ACM, New York (2008)
12. Fernández-Ramil, J., Lozano, A., Wermelinger, M., Capiluppi, A.: Empirical studies of open source evolution. In: Software Evolution, ch. 11, pp. 263–288. Springer, Heidelberg (2008)
13. Fluri, B., Wuersch, M., Pinzger, M., Gall, H.: Change distilling:tree differencing for fine-grained source code change extraction. IEEE Transactions on Software Engineering 33, 725–743 (2007)

14. Jackson, M.: Automated software engineering: supporting understanding. Autom. Softw. Eng. 15(3-4), 275–281 (2008)
15. Kagdi, H.H., Collard, M.L., Maletic, J.I.: A survey and taxonomy of approaches for mining software repositories in the context of software evolution. Journal of Software Maintenance 19(2), 77–131 (2007)
16. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Trans. Softw. Eng. Methodol. 14(3), 331–380 (2005)
17. Madhavji, N.H., Fernandez-Ramil, J., Perry, D.E.: Software Evolution and Feedback: Theory and Practice. Wiley, Chichester (2006)
18. Martin, R.C.: Large-scale stability. C++ Report 9(2), 54–60 (1997)
19. Mens, T., Demeyer, S. (eds.): Software Evolution. Springer, Heidelberg (2008)
20. Osterweil, L.: Software processes are software too. In: Proc. 9th Int'l Conf. on Software Engineering, pp. 2–13. IEEE, Los Alamitos (1987)
21. Popper, K.R.: The Logic of Scientific Discovery. Hutchinson (1959)
22. Schmidt, M., Gloetzner, T.: Constructing difference tools for models using the sidiff framework. In: Proc. 30th Int'l Conf. on Software Engineering, companion volume, pp. 947–948. ACM, New York (2008)
23. Selic, B.: The pragmatics of model-driven development. IEEE Software 20(5), 19–25 (2003)
24. Shull, F., Singer, J., Sjoberg, D.I. (eds.): Guide to Advanced Empirical Software Engineering. Springer, Heidelberg (2008)
25. Stringfellow, C., Amory, C., Potnuri, D., Andrews, A., Georg, M.: Comparison of software architecture reverse engineering methods. Information and Software Technology 48(7), 484–497 (2006)
26. Wermelinger, M., Yu, Y.: Analyzing the evolution of Eclipse plugins. In: Proc. 5th Working Conf. on Mining Software Repositories, pp. 133–136. ACM, New York (May 2008)
27. Wermelinger, M., Yu, Y., Lozano, A.: Design principles in architectural evolution: a case study. In: Proc. 24th Int'l Conf. on Software Maintenance, pp. 396–405. IEEE, Los Alamitos (October 2008)
28. Wermelinger, M., Yu, Y., Strohmaier, M.: Using formal concept analysis to construct and visualise hierarchies of socio-technical relations. In: Proc. 31st Int'l Conf. on Software Eng., companion volume, pp. 327–330. IEEE, Los Alamitos (May 2009)
29. Wong, K.: The Rigi User's Manual, Version 5.4.4 (June 1998)
30. Wu, J., Spitzer, C., Hassan, A., Holt, R.: Evolution spectrographs: visualizing punctuated change in software evolution. In: Proc. 7th Intl Workshop on Principles of Software Evolution, pp. 57–66 (2004)
31. Xing, Z., Stroulia, E.: Umldiff: an algorithm for object-oriented design differencing. In: Proc. 20th Int'l Conf. on Automated Software Engineering, pp. 54–65. ACM, New York (2005)
32. Yu, Y., Wang, Y., Mylopoulos, J., Liaskos, S., Lapouchnian, A., do Prado Leite, J.C.S.: Reverse engineering goal models from legacy code. In: Int'l Conf. on Requirements Engineering, pp. 363–372. IEEE, Los Alamitos (2005)
33. Yu, Y., Wermelinger, M.: Graph-centric tools for understanding the evolution and relationships of software structures. In: Proc. 15th Working Conf. on Reverse Engineering, pp. 329–330. IEEE, Los Alamitos (October 2008)

Teaching Computer Language Handling - From Compiler Theory to Meta-modelling

Terje Gjøsæter and Andreas Prinz

Faculty of Engineering and Science
University of Agder
Serviceboks 509, NO-4898 Grimstad, Norway
{terje.gjosater,andreas.prinz}@uia.no

Abstract. Most universities teach computer language handling by mainly focussing on compiler theory, although MDA (model-driven architecture) and meta-modelling are increasingly important in the software industry as well as in computer science. In this article, we investigate how traditional compiler theory compares to meta-modelling with regard to formally defining the different aspects of a language, and how we can expand the focus in computer language handling courses to also include meta-model-based approaches. We give an outline of a computer language handling course that covers both paradigms, and share some experiences from running a course based on this outline at the University of Agder.

1 Introduction

Although MDA (model-driven architecture) and meta-modelling is increasingly important in the software industry as well as in computer science, many universities still teach language handling with the main focus on compiler theory. For example, in the Norwegian universities, we have found that there is a strong emphasis on compiler theory (CT) and little or no focus on meta-modelling (MM) in most available computer language handling courses (see Table 1).

Compiler theory has traditionally had its strength in defining optimised compilers for large textual general purpose languages. On the other hand, the focus among language designers is shifting towards creating small domain specific languages (DSLs) [1]. These languages may have a graphical or textual presentation (concrete syntax), and they are often based on existing languages and may be preprocessed / embedded / transformed into other languages for execution, instead of being compiled with a traditional compiler.

MDA may have some advantages when it comes to defining these types of languages. An important aspect of MDA is to provide the language designer with support for rapid development and automatic prototyping of language support tools, and allow for working on a high level of abstraction. This approach allows the language designer to focus on the language being developed, while still being able to use the definition for generating tools such as editors, validators and code

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 446–460, 2011.

Table 1. Courses available at Norwegian universities related to computer language handling. The information is collected from course catalogues and course descriptions.

University	Course Name	ECTS	MM	CT	Notes
Bachelor level courses:					
Norwegian U. of Sci. and Tech.	TDT4165 Programming languages	7,5		x	Languages and language implementation
U. of Oslo	INF3110 Programming languages	10		x	Language description
Master level courses:					
U. of Agder	IKT415-C System development with generative programming	5	x	x	Recently revised to also include MM
Norwegian U. of Sci. and Tech.	TDT4205 Compilers	7,5		x	Compiler construction
U. of Oslo	INF5110 Compiler techniques	10	x	x	CT is main focus but MM is mentioned
U. of Bergen	INF225 Introduction to program translation	10		x	Last held in 2005, compiler focussed
No courses currently available:					
U. of Stavanger	N/A				No courses available
U. of Tromsø	N/A				No courses available

generators. Meta-model-based tools are typically based on these principles, but there are also grammar-based tools available that take a similar approach, such as LISA [2].

It may therefore be beneficial to modify university courses in computer language handling to focus not only on compiler development, but also on meta-model-based language design and definition.

The main purpose of this article, is to compare a compiler-theory-based approach with a meta-model-based approach to master level courses in computer language handling, and to examine how that type of courses can be modified from a focus on traditional compiler theory to also cover meta-model-based approaches, tools and technologies. We wish to emphasise that the goal of this paper is not to come up with clear-cut statements about which is better of grammar-based and meta-model-based language definition technologies, but rather to find out which technologies are adequate and suitable for which aspects of a language definition, and how both approaches can be included when teaching computer language handling.

The article is based on literature study, language specifications, and the authors' own experiences with tools, language descriptions as well as teaching of both compiler theory and meta-modelling.

The rest of the article is organised as follows: Section 2 and 3, are introductory sections enumerating the main elements we want to cover in courses in compiler theory and meta-modelling, respectively. Each of these language elements, or language *aspects*, are handled in the following sections; *structure / abstract syntax* in Section 4, *constraints / static semantics* in Section 5, *presentation / concrete*

syntax in Section 6 and *behaviour / dynamic semantics* in Section 7. For each of the language aspects described in sections 4 to 7, we have subsections covering the following topics:

- A general introduction to this aspect.
- Some important issues related to teaching this aspect from the perspective of compiler theory.
- Some important issues related to teaching this aspect from the perspective of meta-modelling, including a selection of meta-model-based tools and technologies that can be used to illustrate the theory of this aspect.
- A comparison of the two approaches from the perspective of teaching this aspect.

In Section 8, we propose an outline of a unified computer language handling course, covering meta-modelling as well as compiler theory. Finally, we summarise our findings in Section 9.

2 A Compiler Theory Curriculum

From Figure 1, we see the basic flow of the main elements of a compiler, and this can also serve as the sequence of main topics for a series of lectures in compiler-construction-based language handling.

Concrete syntax including scanner and parser parts of the compiler, and symbol table generation.
Static semantics including type checking and logical constraints.
Abstract syntax including intermediate code generation and building abstract syntax trees.

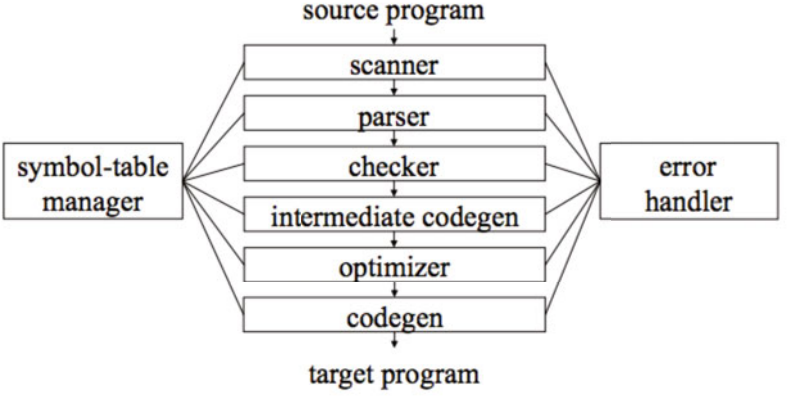


Fig. 1. Elements of a compiler

Translational semantics including optimisation, code generation and error handling.
Execution semantics including run time environments.

In traditional compiler technology, languages are defined by a concrete syntax, an abstract syntax and semantics. Concrete syntax can be precisely defined in BNF/EBNF, and compiler-compiler and parser generator tools like lex and yacc may be used to generate the parser. EBNF can also be used for defining the abstract syntax, however in practice abstract syntax is often automatically derived from the concrete syntax. Although there are well established methods for specifying the formal semantics for a language, in practice, semantics is often not formally defined but developed in an ad-hoc fashion [3].

3 Metamodelling - A Curriculum Based on Aspects of a Programming Language

In [4], a language definition is said to consist of the following aspects: *Structure*, *Constraints*, *Presentation* and *Behaviour* (see Figure 2).

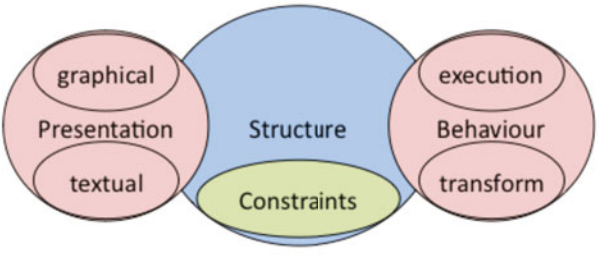


Fig. 2. Aspects of a computer language description

Structure defines the constructs of a language and how they are related.
Constraints bring additional constraints on the structure of the language, beyond what is feasible to express in the structure itself.
Presentation defines how instances of the language are represented. This can be the definition of a graphical or textual concrete language syntax.
Behaviour explains the semantics of the language. This can be a transformation into another language (denotational or translational semantics), or it defines the execution of language instances (operational semantics). Another type of semantics is axiomatic semantics, that gives meaning to phrases of a language by describing the logical axioms that apply to them.

These aspects are not always as strictly separated as they seem in the illustration; constraints are shown as overlapping with structure, since constraints

interact closely with the structure-related technologies in building up (and restricting) the structure of the language. However, constraints can also be used for defining restrictions for presentations as well as behaviour.

The structure is the core of the language; it contains the concepts that should be part of the language, and the relations between them. Traditional grammar-based compiler tools tend to force the focus to the presentation of the language rather than its structure. On the other hand, a meta-model-based approach to language design facilitates a focus on the structure. Starting from a well-defined language structure, it is convenient to define one or more textual and/or graphical presentations for the language, as well as to define code generation into executable target languages such as Java. It is feasible to build a series of lectures in computer language handling on a running example using Eclipse/EMF-based [5] plug-ins and frameworks, to illustrate all aspects of a meta-model-based language definition.

Meta-models define the structure and constraints of a language. For a complete language definition, it is also necessary to define the presentation and behaviour, and relate these definitions to the meta-model, as explained in [3].

Because of this difference in main focus between traditional compiler technology and meta-modelling, it also seems reasonable to let this be reflected in the teaching of these topics. When teaching compiler theory, it is common to start with parsing and grammars, and then later move into abstract syntax and finally semantics and code generation. However, when teaching meta-model-based language design, it is essential to start with teaching how to create a well-formed abstract structure, instead of initially focussing on the presentation of the language. Based on the Structure lecture, should follow lectures on Constraints, Textual and Graphical Presentation, and finally lectures on Transformation (Model-to-Model and Model-to-Text) and Execution.

4 Structure / Abstract Syntax

4.1 Definition

The *structure* of a language specifies what the instances of the language are; it identifies the meaningful components of each language construct [6] and relates them to each other. Based on the language structure definition, a language instance can normally be represented as a tree or a graph. To describe the structure of a computer language therefore means to describe graph structures: what types of nodes exist, and how they can be connected.

There are different levels of expressiveness used in different contexts; grammars, meta-models, database schema descriptions, RDF schemata, and XML schemata are all examples of different ways to express structure.

4.2 Topics and Issues for the Compiler Theory Lecture

Compiler technology commonly uses context-free grammars to define structure. Abstract syntax is in most cases quite similar to the concrete syntax, with some

redundancy removed. Most popular computer languages are grammar-based and do not have a separately described abstract syntax definition, but rely on the concrete syntax.

Grammars can be used to define abstract syntax trees as data structures for language instances [7]. An extended form of context-free grammars are attribute grammars; they define attributed abstract syntax trees. Attributes can help to realise the other aspects of the language: constraints, presentation and behaviour. Attributes can also function as additional connections that turn the abstract syntax tree into a real graph.

The lecture on abstract syntax should include an introduction to grammars and common language structures including regular languages, automata, context free languages, parse trees, abstract syntax trees and attribute grammars.

4.3 Topics and Issues for the Meta-modelling Lecture

While simple grammars define a tree-structure, meta-models are capable of defining a graph. Meta-modelling uses UML's structure modelling constructs to model the structure of languages. A meta-model only defines an abstract structure for a language, because it just describes what the language concepts are and not how their are written or drawn. A meta-model introduces classifiers like classes and associations, for all the constructs in a language. Associations are used to define how instances of these classes, i.e. instances of language constructs, relate to each other.

Meta-modelling allows to modularise, reuse, and combine whole languages or single language constructs. To achieve this, meta-modelling uses object-oriented UML notions like packages and imports, class inheritance, and feature refinement. Object-orientation does not only help with the meta-model design, but also for the design of other definitions and tools based on the meta-model. The graphical nature of meta-models can also facilitate understanding of the structure compared to a textual presentation of structure as is commonly used with grammars.

There are different standards and recommendations for meta-modelling with different complexity and expressiveness. The most famous dialects are MOF 1.x [8], EMF/Ecore [5], and CMOF [9]. The simplicity of EMF/Ecore and EMOF makes it easy to align it to the Java programming language. This fact and EMFs tight integration into Eclipse [10], make it today's most popular meta-modelling language.

Around meta-modelling (especially EMF and Eclipse), many tools and frameworks have been created to easily describe and execute tasks like: persistent language instances in data-bases, validation of language instances, model transformation, execution of language instances, and providing different forms of model editors. Meta-modelling fuelled the vision of creating domain specific languages including comprehensive tools with little resources. Meta-modelling and repositories are the bases for many existing domain specific language and UML case tools.

One weak point with meta-models, is that it is still not well understood what criteria to use to evaluate the quality of a meta-model, and what properties are important and desirable.

4.4 Comparison Related to Teaching

We note that structure can be handled fine with both approaches, but while the structure is usually the starting point when defining a language with meta-model-based technologies, there is less emphasis on this aspect in traditional compiler theory. Therefore, it seems reasonable to start a course in meta-model-based language design with an introduction to structure definition, using for example Eclipse with EMF/Ecore (preferably with a graphical Ecore editor) for demonstrating relevant examples. It should also be noted that the simple tree structures generated from simple grammars are easier to understand for students then the more complex graphs typically formed by meta-models and attribute grammars.

5 Constraints / Static Semantics

5.1 Definition

Constraints on a language can put limitations on the structure of a well-formed instance of the language. This aspect of a language definition mostly concerns logical rules or constraints on the structure that are difficult to express directly in the structure itself. Neither meta-models nor grammars provide all the expressiveness that is needed to define the set of wanted language instances. The constraints could for example be first-order logical constraints or multiplicity constraints for elements of the structure [11].

There is an overlap between the structure and constraint aspects of a language. Some language features may obviously belong to one of them, but many features could belong to either of them, depending on choice or on the expressiveness of the technology used to define the structure.

5.2 Topics and Issues for the Compiler Theory Lecture

What we want to express here are logical rules (static semantic conditions) related to elements of the language structure. Often, these constraints are expressed in code, and in some cases in a logic language. These logical rules may be attached to attributes in an attribute grammar. This lecture should also include an introduction to type systems and type checking.

5.3 Topics and Issues for the Meta-modelling Lecture

While meta-models are constructive definitions of what objects a language instance can consist of, constraints allow to narrow down the possible instances of a meta-model class. A meta-model constraint is thereby always written in the

context of a class, and only constrains the set of possible instances (objects) of this class. A constraint forms a logical expression. It takes an instance of the context class as input and evaluates to a boolean value, assessing the instance as either a valid, or an invalid instance. Only the models that exclusively consist of valid objects are valid language instances.

In meta-modelling, the most common technology for expressing constraints is the Object Constraint Language, OCL. OCL is designed to present the expressiveness of predicate logic, in a programming language like syntax. Related tools allow to check whole models or single objects, based on the constraints associated with the model's meta-model classes. Language tools based on meta-models, usually do not check a model within a separate tool, but are integrated into model editors. Model editors check single objects and can display invalid objects to the user.

5.4 Comparison Related to Teaching

We see that constraints can be handled fine with both approaches. There is often more emphasis on explicitly defined constraints in meta-model-based development. Teaching constraints will fit naturally as an extension to lectures about structure, and can be illustrated by creating and adding logical expressions to an example grammar and OCL constraints to an example meta-model.

6 Presentation / Concrete Syntax

6.1 Definition

The *presentation* of a language describes the possible forms of a statement of the language. In the case of a textual language, it describes what words are allowed to use in the language, what words have special meaning and are reserved, and what words are possible to use for variable names. It may also describe what sequence the elements of the language may occur in; the syntactic features of the language. This is expressed in a grammar for textual languages.

Similarly, in a graphical language, the presentation will express what different symbols are used in the language, and how they can be connected and modified to form a meaningful unit in the language. The *presentation of graphical languages* can be defined in two ways:

The "constructive" way is generator-based, using graph grammars.
The "direct" way may describe a model for the graph.

In addition to defining the graph structure, we may wish to define attributes such as location, shape, and colour of the different elements in the graph.

Describing language structures separately from the language's notation(s), allows us to define multiple notations for the same language, and allows for arbitrary kinds of notations, graphical (diagrams and other variants) as well as textual.

There are two major ways to *connect the presentation to the structure* of a language;

The "constructive" way is done by defining a transformation between presentation and structure.

The "abstract" way is based on pattern matching, showing how elements of the presentation are connected to the structure elements.

We have two main approaches to creating tools for handling presentation of a language;

Parsers that have to support a one-way connection from the presentation to the corresponding structure.

Editors that have to support a two-way connection between the presentation and the corresponding structure, providing feedback from the syntax analysis in form of syntax highlighting, error messages, code completion suggestions etc.

In addition, executable output text in the form of machine code or byte code can also be considered a presentation of a language instance. Code generators have to support a one-way connection from the structure to a presentation of the code to be generated.

6.2 Topics and Issues for the Compiler Theory Lecture

The presentation of a language is in traditional compiler theory called *concrete syntax*. Context free grammars for the concrete syntax of a programming language are often written in BNF (Backus–Naur form) or EBNF (Extended BNF) notation, and most popular parser generators use grammars based on an (E)BNF-like syntax.

This lecture should include a basic introduction to different grammar types such as LL, LR, SLR, LALR; and also parse tables, canonical sets, first-follow sets, conflicts (shift-reduce and reduce-reduce), conflict resolution, and mapping between concrete and abstract syntax. For graphical concrete syntax, a brief introduction to graph grammars should be included. Symbol table management and error handling may also fit into this lecture.

6.3 Topics and Issues for the Meta-modelling Lecture

Meta-models describe language structures with classes and associations resulting in language instances that are graphs, rather than the tree structures normally generated by grammars. Therefore, a meta-model is a suitable basis for defining graphical notations.

In general, notations are described in a separate formalism. Textual notation for example can be described with context-free grammars, graphical notations can be described in their own meta-model for shapes and connections. A third model then defines a mapping between the meta-model elements and the notation definition's elements.

A formalism to define a certain kind of notation consist of a notation definition language and a mapping definition language. Existing formalism for notations definition are usually embedded in frameworks and tools that realise

them. Existing formalisms for graphical notations and textual notations allow to automatically create graphical editors and feature-rich text editors, including error annotations (for validations), code-completion, name-resolution, syntax highlighting, etc.

Graph grammars have also been suggested for use in creating modelling tools. See for example [12] for more information on graph grammars.

Frameworks for textual notations can be divided into tools like XText [13], which actually provides editors solely based on language definitions consisting of grammars, and frameworks like TCS [14], TEF [15] and EMFText [16], which combine meta-models and grammars. XText allows to define a language syntax and implicitly a language structure based on a grammar-like definition. XText generates a meta-model and a textual editor for this meta-model from this definition. The editor continuously generates a meta-model instance, by parsing the text entered by the user. The other frameworks allow to provide a grammar and grammar-to-meta-model mapping based on already existing meta-models. They generate editors that allow creation of meta-model instances, by parsing the user text. These frameworks use techniques similar to those of attribute grammars to handle non-containment model structures and provide automatic support for resolving named references based on these techniques.

One well-known framework for graphical notations is GMF [17]. It features a language to define graphical notations, including different shapes, shape containment, connections, and labels for different elements. GMF allows to define simple mappings between meta-model elements and the elements of a graphical notation. GMF generates Eclipse and GEF-based [18] editors from these definitions. This is fully functional for simple language notations, and can be enriched by manually altering the generated code.

6.4 Comparison Related to Teaching

While textual presentation may be a natural starting point in compiler theory based teaching, it is more suitable to let lectures on presentation build on a foundation of basic structure. In this part, the students should get practice in defining grammars for simple languages as well as deriving languages from grammars. If a running meta-model-based example is used, it may be fruitful to show the students how an EMF-based example structure (with constraints) can be extended with both graphical and textual presentations, using editor generation frameworks like for example GMF for graphical editor generation and EMFText for textual editor generation.

7 Behaviour / Dynamic Semantics

7.1 Definition

The *behaviour* of a language describes what is the actual meaning of a statement of the language.

Two main types of formal ways of defining semantics are called operational and denotational semantics [6]:

Denotational semantics in the strict sense, is a mapping of a source expression to an input-output function working on some mathematical entities. If we wish to include model transformations and language-to-language translations in our behaviour descriptions, we can include them in this category by applying a more broad definition of denotational semantics; namely a transformation of each phrase of the language into a phrase in some other language, often a mathematical formalism. To execute or interpret the behaviour of a statement, semantics for the target language is then needed. A denotational semantics describes an "abstract" compiler.

Operational semantics describes the execution of the language as a sequence of computational steps. You will then need to know the semantics of the interpreter. Operational semantics may be described by state transitions for an abstract machine. In [11], it is described how semantics for SDL are handled by Abstract State Machines (ASM). With operational semantics, a runtime environment is needed. An operational semantics describes an "abstract" interpreter.

A third type of semantics, *Axiomatic* semantics, gives meaning to phrases of a language by describing the logical axioms that apply to them. Experience shows that axiomatic semantics are extremely complex and rarely used for computer languages. For this paper we only focus on denotational and operational semantics.

7.2 Topics and Issues for the Compiler Theory Lecture

Semantics has traditionally been an area that is much less formalised than the structure or abstract syntax of a language. In most cases, the semantics has been described in plain English, or by reference implementation of a compiler or interpreter for the language. However, for attribute grammars, it is quite common to attach more formal semantic rules to the attributes.

7.3 Topics and Issues for the Meta-modelling Lecture

Similar to grammars, in many cases, the semantics has been described in plain English, or by reference implementation.

While the focus in traditional compiler theory teaching often is on code generation, it is natural in a course focussing on meta-model-based technologies to cover model-to-model transformations as well as model-to-text transformations. For the former, transformation languages like QVT or ATL can be used to create example transformations on the structure of the running EMF-based example, and for the latter, JET [19], Acceleo [20] or XPand [13] can be used to generate textual code.

There is plenty of academic work that suggests the definition of operational semantics based on a meta-model. These approaches create state-transitions systems to describe behaviour along Plotkin's classical operational semantics [21]. These systems are based on meta-models as a definition for the set of states, and depending on the approach use transformations based on graph transformations

[22], or some form of action languages, like UML Activities [23], or ASM [11] to define possible state transitions. The Eclipse plugin EProvide [24], provides support for developing visual debuggers and interpreters based on operational semantics defined in ASM, QVT/Relations, Java, Prolog or Scheme.

7.4 Comparison Related to Teaching

We have noted that it may be challenging to teach this language aspect since most of the tools available for supporting the theory of this aspect are relatively immature and/or hard to use, particularly for execution behaviour. Model-to-model transformations tend to be better supported by meta-model-based tools and technologies. Model-to-text transformation is adequately supported by both approaches. On the other hand, execution is not well supported in any of the two approaches.

For illustrating the theory in this lecture, we may want to give the structure of our running example Model-to-Model transformations using QVT or ATL, and Model-to-Text with for example JET or XPand. It may also be useful to demonstrate operational semantics with ASM-based semantics in EProvide.

8 A Computer Language Handling Course Outline

From the ideas developed in the previous sections, we have defined the following course outline:

Level: MSc.
Prerequisites: Object oriented programming, UML modelling.
Credits: 5 ECTS.
Literature: Aho, Lam, Sethi, Ullman: Compilers (2nd ed.)[25]; Clark, Sammut, Willans et. al.: Applied Metamodeling (2nd ed.) [26].
Form: 7 parts; each part with lectures, practical and theoretical exercises, and an obligatory hand-in.

Part 1 - Introduction: Compilers, languages, language aspects, grammars, NFA and DFA automata, T-diagrams.
Part 2 - Structure: Models, meta-models, MDA, EMF/Ecore, abstract syntax, attribute grammars.
Part 3 - Constraints: Semantic analysis, type systems, static and dynamic checks, type safety, logical constraints, OCL.
Part 4 - Textual presentation: Syntax analysis, top-down and bottom-up parsing, lexical analysis, mapping, symbol tables, error handling, TEF, EMF-Text.
Part 5 - Graphical presentation: Graphical languages, graph grammars, GMF.
Part 6 - Transformation behaviour: Transformation, code generation, intermediate code, optimisation, handling of generated code, JET, QVT.

Part 7 - Execution behaviour: Semantics, interpreters, runtime environments, storage allocation, activation records, parameter passing, dynamic binding, ASM, EProvide.
Part 8 - Summary: Repetition of the most important topics of the course.

In the related project course, the students have a choice of different projects building on this course.

8.1 Experiences

The course has been implemented at the University of Agder in the spring term of 2010. After running the course, the following experiences were gathered:

- It is good to use a running example where aspects are added to complete a simple example language. It is also beneficial to cover all language aspects within one platform. However, students can easily be demotivated by immature tools.
- We should not try to cover too many different tools in the practical exercises, but rather concentrate on the most important ones and give the students more time to try them out for themselves by modifying and extending provided examples.
- The understanding should be strengthened by giving different perspectives on the same issues in a lecture covering both compiler theory and meta-modelling. However, the connection between the two paradigms were sometimes difficult for the students to see.

9 Conclusions

When it comes to teaching of computer language handling, we conclude that although traditional compiler theory should still play an important role, there is also a need for a stronger focus on meta-model based technologies. Although the two paradigms are emphasising different aspects of language definition, we have shown that it is still possible to cover all important language aspects relatively well with either paradigm.

It is possible to build a series of lectures in computer language handling where both compiler-theory-based and meta-model-based approaches are covered. The meta-model-based approach can be illustrated by running examples based on Eclipse/EMF and other Eclipse-based plug-ins and frameworks, to cover all aspects of a language definition.

We have found that it is essential to emphasise the connections and similarities between compiler-theory-based and meta-model-based approaches to language handling, and to avoid the least mature tools. It is also important to ensure that the students has a common software platform to lower the risks of unexpected bugs and problems.

References

1. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling. Wiley Interscience, Hoboken (2008)
2. Henriques, P.R., Pereira, M.J., Mernik, M., Lenič, M.: Automatic generation of language-based tools. In: Second Workshop on Language Descriptions, Tools and Applications, LDTA 2002. Electronic Notes in Theoretical Computer Science, vol. 65, pp. 77–96. Elsevier Science Publishers, Amsterdam (2002)
3. Kleppe, A.: A language is more than a metamodel. In: ATEM 2007 Workshop (2007), http://megaplanet.org/atem2007/ATEM2007-18.pdf
4. Nytun, J.P., Prinz, A., Tveit, M.S.: Automatic generation of modelling tools. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 268–283. Springer, Heidelberg (2006)
5. Griffin, C.: Using EMF. Technical report, IBM: Eclipse Corner Article (2003), http://www.eclipse.org/articles/Article-UsingEMF/using-emf.html
6. Sethi, R.: Programming Languages Concepts and Constructs. Addison-Wesley, Reading (1996)
7. van Eijk, P., Belinfante, A., Eertink, H., Alblas, H.: The Term Processor Generator Kimwitu. CTIT Technical report 96-49, University of Twente (1996), http://fmt.cs.utwente.nl/kimwitu/
8. OMG Editor: Meta Object Facility (MOF) Specification. Technical report, Object Management Group (2002), http://www.omg.org/docs/formal/02-04-03.pdf
9. OMG Editor: Revised Submission to OMG RFP ad/2003-04-07: Meta Object Facility (MOF) 2.0 Core Proposal. Technical report, Object Management Group (2003), http://www.omg.org/docs/formal/06-01-01.pdf
10. d'Anjou, J., Fairbrother, S., Kehn, D., Kellermann, J., McCarthy, P.: The Java Developer's Guide to Eclipse. Addison-Wesley, Reading (2004)
11. Prinz, A., Scheidgen, M., Tveit, M.S.: A Model-based Standard for SDL. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. 4745, pp. 1–18. Springer, Heidelberg (2007)
12. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation. Applications, Languages and tools, vol. 2. World Scientific, Singapore (1999)
13. Efftinge, S., Friese, P., Haase, A., Hübner, D., Kadura, C., Kolb, B., Köhnlein, J., Moroff, D., Thoms, K., Völter, M., Schönbach, P., Eysholdt, M.: OpenArchitectureWare User Guide (2008), http://www.eclipse.org/gmt/oaw/doc/4.3/html/contents/index.html
14. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: Proceedings of the Fifth International Conference on Generative Programming and Component Engineering, GPCE 2006, pp. 249–254 (2006)
15. Scheidgen, M.: Textual Editing Framework (2008), http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/documentation.html
16. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and refinement of textual syntax for models. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 114–129. Springer, Heidelberg (2009)
17. GMF developers: Eclipse Graphical Modeling Framework (2008), http://www.eclipse.org/gmf

18. GEF developers: GEF documentation (2008), http://www.eclipse.org/gef/reference/documentation.php
19. JET developers: JET Tutorial part 1 (2004), http://www.eclipse.org/articles/ArticleJET/jet_tutorial1.html
20. Musset, J., Juliot, É., Lacrampe, S.: Acceleo User Guide (2008), http://acceleo.org/doc/obeo/en/acceleo-2.6-user-guide.pdf.2.6edn
21. Plotkin, G.D.: A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, University of Aarhus (1981)
22. Grunske, L., Geiger, L., Zündorf, A., Eetvelde, V., Van Gorp Niels, P., Varró, D.: Using Graph Transformation for Practical Model Driven Software Engineering. In: Model Driven Software Engineering, pp. 91–118. Springer, Heidelberg (2005)
23. Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530, pp. 157–171. Springer, Heidelberg (2007)
24. Sadilek, D.A., Wachsmuth, G.: Prototyping visual interpreters and debuggers for domain-specific modelling languages. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 63–78. Springer, Heidelberg (2008)
25. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley, Reading (2007)
26. Clark, T., Sammut, P., Willans, J.: Applied Metamodeling – A Foundation for Language Driven Development, 2nd edn. Ceteva (2008)

C++ Metastring Library and Its Applications*

Zalán Szűgyi, Ábel Sinkovics, Norbert Pataki, and Zoltán Porkoláb

Department of Programming Languages and Compilers, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
{lupin,abel,patakino,gsd}@elte.hu

Abstract. C++ template metaprogramming is an emerging direction of generative programming: with proper template definitions we can enforce the C++ compiler to execute algorithms at compilation time. Template metaprograms have become essential part of today's C++ programs of industrial size; they provide code adoptions, various optimizations, DSL embedding, etc. Besides the compilation time algorithms, template metaprogram data-structures are particularly important. From simple typelists to more advanced STL-like data types there are a variety of such constructs. Interesting enough, until recently string, as one of the most widely used data type of programming, has not been supported. Although, `boost::mpl::string` is an advance in this area, it still lacks the most fundamental string operations. In this paper, we analysed the possibilities of handling string objects at compilation time with a metastring library. We created a C++ template metaprogram library that provides the common string operations, like creating sub-strings, concatenation, replace, and similar. To provide real-life use-cases we implemented two applications on top of our Metastring library. One use case utilizes compilation time inspection of input in the domain of pattern matching algorithms, thus we are able to select the more optimal search method at compilation time. The other use-case implements `safePrint`, a type-safe version of `printf` – a widely investigated problem. We present both the performance improvements and extended functionality we have achieved in the applications of our Metastring library.

1 Introduction

Generative programming is an emerging programming paradigm. The C++ programming language [22] supports the generative programming paradigm with using the *template* facility. Templates are designed to shift the classes and algorithms to a higher abstraction level without losing efficiency. They enable data structures and algorithms to be parametrised by types, thus, the classes and algorithms can be more general and flexible. It makes the source code shorter and easier to read and maintain which improves the quality of the code. Templates and template-based libraries – most notably the Standard Template Library (STL) – is now an unavoidable part of professional C++ programs.

* Supported by TÁMOP-4.2.1/B-09/1/KMR-2010-0003.

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 461–480, 2011.

C++ templates – as opposed to the Java and C# solution – work using the *instantiation* mechanism. Instantiation happens when a template is referred to with some concrete arguments. During instantiation the template parameters are substituted with the concrete arguments and the generated code is compiled.

The instantiation mechanism has an – originally unintentional – side effect. By defining clever template constructs we can enforce the C++ compiler to execute algorithms at compilation time. To demonstrate this in 1994 Erwin Unruh wrote a program which printed a list of prime numbers as part of error messages [23]. Unruh used template definitions and template instantiation rules to compute the primes at compilation time. This programming style is called C++ template metaprogramming [26]. The template metaprogram itself "runs" at compilation time. The output of this process is the generated C++ code – in most cases not the pure source code, but its internal representation – which is also checked by the compiler. The generated program can run as an ordinary "run-time" program. Template metaprogramming has been proven to be a Turing-complete sub-language of C++ [5].

Template metaprogramming is widely used today for several purposes, like executing algorithms at compilation time to optimize or make safer run-time algorithms and data structures. *Expression templates* were the first applications [25] allowing C++ expressions to be evaluated lazily and eliminating the overhead of object-oriented programming mainly in numerical computations.

Static interface checking increases the safety of the code, allowing checking at compilation time whether template parameters meet the given requirements [19]. As the C++ programming language has no language support to describe explicit requirements for certain template properties, only the template metaprogram based library solutions [29] remain.

The classical compilation model of software development designs and implements sub-programs, then compiles them and runs the program. During the first step the programmer makes decisions about implementation details: choosing algorithms, setting parameters and constants. Using template metaprograms some of these decisions can be delayed. *Active libraries* [24,13] take effect at compilation time, making decisions based on programming contexts. In contrast to traditional libraries they are not passive collections of routines or objects, but take an active role in generating code. Active libraries provide higher abstractions and can optimize those abstractions themselves.

Domain specific languages (DSLs) are dedicated to special problems. They are often incorporated into some general purpose host language – many times into C++. The *Ararat* system [9], *boost::xpressive* and *boost::proto* [38,32] libraries are good examples to libraries for embedding DSLs.

In the last fifteen years lots of research activities focused on improving the process of metaprogramming. Essential compilation time algorithms have been identified and used to develop basic metaprogram libraries [1,2]. Complex data structures are also available for metaprograms. Recursive templates store

information in various forms, most frequently as lists or tree structures. The canonical examples for sequential data structures are `typelist` [1] and the elements of the `boost::mpl` library [30].

Strings are one of the most commonly used data types in programming. Some programming languages provide strings as built-in data types, while others support strings and their operations by their standard library. A number of applications are based on string manipulation, like lexical analysers, pattern matchers and serialization tools. These applications are widely used in most areas of computer science. Numerous research activities and studies managed to improve the efficiency of these algorithms, however these improvements focused only on run time algorithm optimizations.

Sometimes, part of the input arguments of string manipulation algorithms are known at compilation time. In these cases a template metaprogram is able to customize the string algorithm to the corresponding input, making it safer and more efficient. While using the *Knuth-Morris-Pratt* sub-string search algorithm [4] we know the exact pattern we are searching in the text. Thus, we can generate the *next* vector of the algorithm at compilation time. As an other example the *regular expression* library `boost:xpressive` is able to check the syntax of the matching pattern at compilation time to detect erroneous regular expressions.

As more and more complex applications of template metaprogramming have appeared, it is surprising that for a long time strings were not supported for compilation time programming. The first attempt, `boost::mpl::string` [34] has been created recently, and still lacks a number of essential features like compare, search and replace sub-strings. Therefore we extended `boost::mpl::string` to create our own metastring library to provide a better support for string manipulation in metaprograms. In this paper, we present the meta-algorithms of usual string operations.

To illustrate the importance of the Metastring library, we demonstrate its usage by detailed use cases. One of the examples is from the searching algorithm domain. Knowing the pattern to search at compilation time, we are able to choose various optimizations to improve our algorithm.

The other example is the implementation of the `printf` C function in a type-safe way. Although the `printf` function of the C standard library has a compact and practical syntax – that is why `printf` is so widely used even today – it is not recommended to use it in C++, because it parses the formatting string at run time, thus, it is not type-safe. At the same time, in most cases the formatter string is already known at compilation time. Hence, it is possible to generate a formatter string specific `printf` using metaprograms, so that the compiler is able to check the type of the parameters, making the code safer. We show that our type-safe `printf` performs better than the type-safe stream operations from the standard C++ library.

The paper is organized as follows. In Section 2 we give a short description of the template metaprogramming techniques. Section 3 introduces our Metastring library. In Section 4 we discuss our pattern matching improvements as

applications of the metastring construct. In Section 5 we present the type-safe `printf`. We give an overview on related and future works in Section 6, and we summarize our results in Section 7.

2 Template Metaprograms

The template facility of C++ allows writing algorithms and data structures parametrized by types. This abstraction is useful for designing general algorithms like finding an element in a list. The operations of lists of integers, characters or even user defined classes are essentially the same. The only difference between them is the stored type. With templates we can parametrize these list operations by type, thus, we need to write the abstract algorithm only once. The compiler will generate the integer, double, character or user defined class version of the list from it. See the example below:

```
template<typename T>
struct list
{
  void insert(const T& t);
  // ...
};

int main()
{
  list<int> l;     //instantiation for int
  list<double> d;  //instantiation for double
  l.insert(42); d.insert(3.14);  // usage
}
```

The list type has one template argument `T`. This refers to the parameter type, whose objects will be contained in the list. To use this list we need to generate an instance and assign a specific type to it. That method is called *instantiation*. During this process the compiler replaces the abstract type `T` with a specific type and compiles this newly generated code. The instantiation can be invoked either explicitly by the programmer but in most cases it is done implicitly by the compiler when the new list is first referred to.

The template mechanism of C++ enables the definition of partial and full *specializations*. Let us suppose that we would like to create a more space efficient type-specific implementation of the `list` template for `bool` type. We may define the following specialization:

```
template<>
struct list<bool>
{
  //type-specific implementation
};
```

Nevertheless, the implementation of the specialized version can be totally different from the original one. Only the names of these template types are the same. If during the instantiation the concrete type argument is `bool`, the specific version of `list<bool>` is chosen, otherwise the general one is selected.

Template specialization is essential practice for template metaprogramming too. In template metaprograms templates usually refer to other templates, sometimes from the same class with different type argument. In this situation an implicit instantiation will be performed. Such chains of recursive instantiations can be terminated by a template specialization. See the following example of calculating the factorial value of 5:

```
template<int N>
struct factorial
{
  enum { value = N * factorial<N-1>::value };
};

template<>
struct factorial<0>
{
  enum { value = 1 };
};

int main()
{
  int result = factorial<5>::value;
}
```

To initialize the variable `result` here, the expression `factorial<5>::value` has to be evaluated. As the template argument is not zero, the compiler instantiates the general version of the `factorial` template with 5. The definition of `value` is `N * factorial<N-1>::value`, hence the compiler has to instantiate the `factorial` again with 4. This chain continues until the concrete value becomes 0. Then, the compiler choses the special version of `factorial` where the `value` is 1. Thus, the instantiation chain is stopped and the factorial of 5 is calculated and used as initial value of the `result` variable in `main`. This metaprogram "runs" while the compiler compiles the code.

Template metaprograms therefore stand for the collection of templates, their instantiations and specializations, and perform operations at compilation time. The basic control structures like iteration and condition appear in them in a functional way [20]. As we can see in the previous example, iterations in metaprograms are applied by recursion. Besides, the condition is implemented by a template structure and its specialization.

```
template<bool cond_, typename then_, typename else_>
struct if_
{
  typedef then_ type;
};

template<typename then_, typename else_>
struct if_<false, then_, else_>
{
  typedef else_ type;
};
```

The `if_` structure has three template arguments: a boolean and two abstract types. If the `cond_` is false, then the partly-specialized version of `if_` will be instantiated, thus the `type` will be bound by the `else_`. Otherwise the general version of `if_` will be instantiated and `type` will be bound by `then_`.

Complex data structures are also available for metaprograms. Recursive templates store information in various forms, most frequently as tree structures, or sequences. Tree structures are the favorite forms of implementation of expression templates [25]. The canonical examples for sequential data structures are `typelist` [1] and the elements of the `boost::mpl` library [30].

We define a typelist with the following recursive template:

```
class NullType {};
struct EmptyType {};        // could be instantiated

template <typename H, typename T>
struct Typelist
{
  typedef H head;
  typedef T tail;
};
typedef Typelist< char, Typelist<signed char,
             Typelist<unsigned char, NullType> > > Charlist;
```

In the example we store the three character types in a typelist. We can use helper macro definitions to make the syntax more readable.

```
#define TYPELIST_1(x)           Typelist< x, NullType>
#define TYPELIST_2(x, y)        Typelist< x, TYPELIST_1(y)>
#define TYPELIST_3(x, y, z)     Typelist< x, TYPELIST_2(y,z)>
// ...
typedef TYPELIST_3(char, signed char, unsigned char)  Charlist;
```

Essential helper functions – like `Length`, which computes the size of a list at compilation time – have been defined in Alexandrescu's Loki library[1] in pure functional programming style. Similar data structures and algorithms can be found in the `boost::mpl` metaprogramming library [30].

3 Metastring Library

In this chapter we introduce our *metastring* library. In the examples we write the type- and function names of boost without the scope (`string`, instead of `boost::mpl::string`) to save space. If we write the names of functions or objects in STL we put the scope before them.

Metastring library is based on `boost::mpl::string` [34]. The Boost Metaprogram Library provides us a variety of meta containers, meta algorithms and meta iterators. The design of that library is based on STL, the standard library of C++. However, while the STL acts at run time, the `boost::mpl` works at compilation time. The meta version of regular containers in STL, like list, vector, deque, set and map are provided by `boost::mpl`. Also, there are meta versions of most algorithms and iterators. The string metatype was added to boost in the release 1.40. Contrary to other meta containers, the metastring has limited features. Almost all regular string operations, like concatenation, equality comparison, substring selection, etc. are missing. Only the `c_str` meta function, which converts the metastring type to constant character array, is provided by boost.

In our Metastring library we extended the `boost::mpl::string` with the most common string operations. The `boost::mpl::string` is a *variadic template* type [10] like the `boost::mpl::vector` [36], but only accepts characters as template arguments. The instantiated metastring type can perform as a concrete string at compilation time. Since an instantiated metastring is a type, one can assign a shorter name to it by the `typedef` keyword.

Certain programming languages define the string datatype as a sequence of characters; i.e. array or list of characters. The metastring itself is a template type. The template arguments contain the value of the string. Because C++ does not support passing string literals to template arguments [18], we need to pass string arguments character by character.

```
typedef string<'H','e','l','l','o'> str;
```

Setting metastrings char-by-char is very inconvenient, therefore, the boost library offers an improvement. In most architectures, the `int` contains at least four bytes and since the size of a character is one byte, it can store four characters. As a template argument can be any integral type, hence it is possible to pass four characters as integer, and later on a metaprogram transforms it back to characters. This provides a more readable notation:

```
typedef string<'Hell','o'> str;
```

Nevertheless, this is still not the simplest way of setting a metastring. The next C++ standard will provide a better and standard solution: with the combination of *variadic templates* [10] and *user defined literals* [28] we can pass a string in the form of `"Hello"s` to variadic templates with character arguments.

Since the literature usually uses the parameter passing by four character convention, in the rest of the paper we will follow that.

In the Metastring library we provide the meta-algorithms of the most common string operations, like `concat`, `find`, `substr`, `equal`, etc. These algorithms are also template types, which accept metastring types as template arguments. The `concat` and `substr` defines a type called `type` which is the result of the operation. `equals` provides a static boolean constant called `value`, which is initialized as true if the two strings are equal, otherwise as false. `find` defines a static `std::size_t` constant called `value`, which is initialized as the first index of matching, if the pattern appears in the text and as std::string::npos otherwise. See the example about concatenation of strings below:

```
typedef string<'Hell','o'> str1;
typedef string<' Wor','ld!'> str2;

typedef concat<str1, str2>::type res;

std::cout << c_str<res>::value
```

The type defined by `concat<str1, str2>` is a new metastring type, which represents the concatenation of `str1` and `str2` metastrings.

In the next chapter we present applications which can take either efficiency or safety advantages of metastrings. The first example is pattern matching. If the text or a pattern is known at compilation time, we can improve the matching algorithms. (If both the text and the pattern are known, we can perform the whole pattern matching algorithm at compilation time.) The second application is a type-safe `printf`. If the formatter string is known at compilation time, we can generate a specialized kind of `printf` algorithm to it, which can perform type checking.

4 Pattern Matching Applications with Metastrings

Most of the pattern matching algorithms start with an initialization step. This step depends only on the pattern. If the pattern is known at compilation time, we can shift this initialization subroutine from run time to compilation time. This means that while the compiler compiles the code it will wire the result of the initialization subroutine into the code. Thus, the algorithm does not need to run the initialization step, because it is already initialized. The more often the pattern matching algorithm is invoked, the more speed-up we achieve. The example below shows how to use these algorithms:

```
typedef string<'patt','ern'> pattern;
std::string text;
// reading data to text

std::size_t res1 = kmp<pattern>(text);
std::size_t res2 = bm<pattern>(text);
```

The `kmp` and `bm` function templates implement the Knuth-Morris-Pratt [15] and the Boyer-Moore [3] pattern matching algorithms. The return values are similar to the `std::string`'s `find` memberfunction in `STL` and are either the first index of the match or `std::string::npos`. The implementation of these functions are the following:

```
template<typename pattern>
std::size_t kmp(const std::string& text)
{
  const char* p = c_str<pattern>::value;
  const char* next = c_str<initnext<pattern>::type>::value;
  //implementation of Knuth-Morris-Pratt
}

template<typename pattern>
std::size_t bm(const std::string& text)
{
  const char* pattern = c_str<pattern>::value;
  const char* skip = c_str<initskip<pattern>::type>::value;
  //implementation of Boyer-Moore
}
```

The `pattern` template argument must be a metastring type for both of the functions. The `initnext` and the `initskip` meta algorithms create the `next` and the `skip` vectors for the algorithms at compilation time. The rest of the algorithms are the same as the normal run time version.

We compared the full run time version of algorithms with our solution where the initialization is performed at compilation time. Fig. 1 shows the results related to Knuth-Morris-Pratt and fig. 2 to Boyer-Moore. We tested these algorithms with several inputs. The input was a common English text and the pattern contained a couple of words. The pattern did not appear in the text, thus the algorithms had to read all the input. We measured the running cost with one, two, five and ten kilobyte long inputs. In both charts, the first columns show the running cost of the original algorithms and the second ones show the performance of the algorithms optimized at compilation time. The X-axis shows the inputs and the Y-axis shows the instructions consumed during the algorithm. The larger improvement can be achieved applying pattern match repeatedly.

It is quite rare but still interesting case when the text is known in compile time. In this case a meta program can analyze the characteristic of the text and choses the best pattern matching algorithm. For example if the alphabet of the input text is large, the Boyer-Moore is more efficient, but if the alphabet is small, choosing the Knuth-Morris-Pratt algorithm is more beneficial. The figure 3 shows the differences of search speed, when a search is performed by only Knuth-Morris-Pratt (1st group) or Boyer-Moore(2nd group) algorithm or the optimized version (3rd group). `InputA` denotes an ordinary English text and the

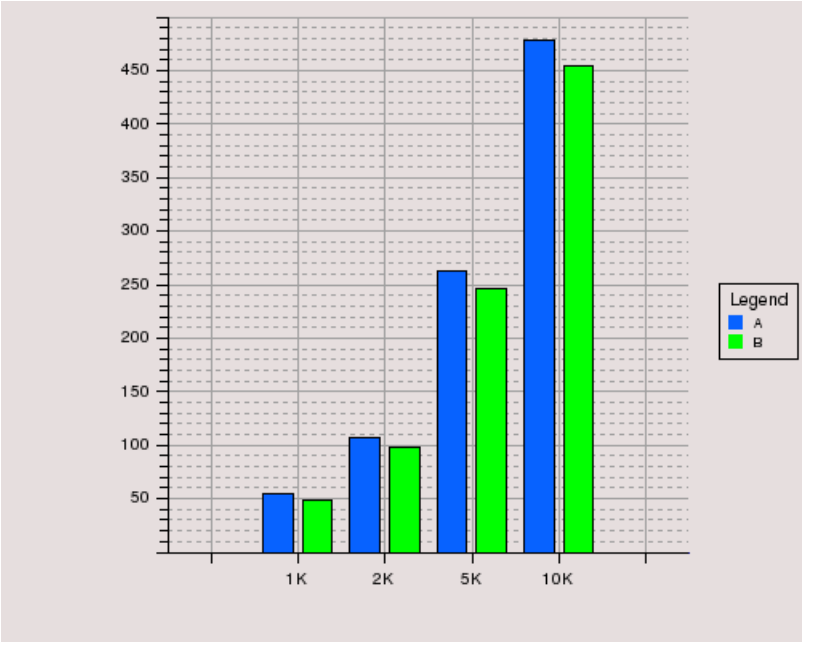


Fig. 1. Comparison of Knuth-Morris-Pratt

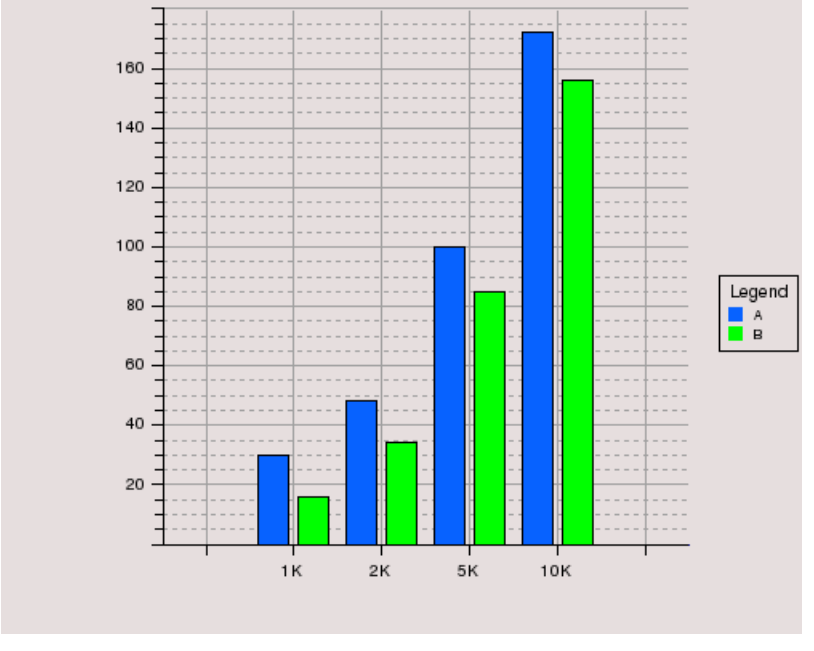


Fig. 2. Comparison of Boyer-Moore

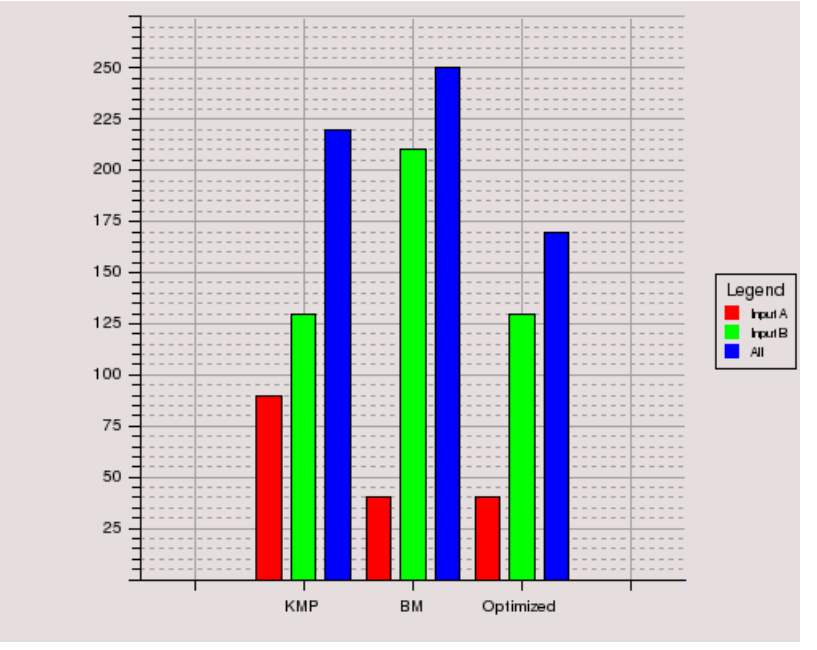


Fig. 3. Comparison of algorithms

pattern is a single word. `InputB` denotes the text containing a few characters, and the pattern has several repetitions. The Y-axis illustrates the consumed instructions.

5 Type-Safe Printf

The `printf` function of the standard C library is easy to use and efficient but has a major drawback: it is not type-safe. Due to the lack of type-safety, mistakes of the programmer may cause undefined behavior at runtime, because the compiler does not verify the validity of the arguments passed to `printf`. There are workarounds, for example gcc type checks `printf` calls and emits warnings when they are incorrect, but it is specific to gcc. C++ introduced *iostreams* as a replacement of `printf`. Iostreams are type-safe, but they have runtime and syntactical overhead. The syntax of `printf` is more compact than the syntax of streams, the structure of the displayed message is defined at one place, in the *format string*, when we use `printf` but it is scattered across the whole expression when we use streams. Here is an example for using `printf` and streams to display the same thing:

```
printf("Name: %s, Age: %d\n", name, age);
std::cout << "Name: " << name << ", Age: " << age << std::endl;
```

In this section we implement a type-safe version of `printf` using compilation time strings assuming that the format string is available at compilation time, which is true in most cases. We write a C++ wrapper for `printf` which validates the number and type of its arguments at compilation time and calls the original `printf` without any runtime overhead.

We call the type-safe replacement of `printf` `safePrintf`. It is a template function taking one class as a template argument: the format string as a compilation time string. The arguments of the function are the arguments passed to `printf`. As the example usage

```
safePrintf< string<'Hell', 'o %s', '!'> >("John");
```

shows there is only a slight difference between the usage of `printf` and our type-safe `safePrintf`. On the other hand, there is a significant difference between their safety: `safePrintf` guarantees that the `printf` function called at runtime has the right number of arguments and they have the right type.

Under the hood `safePrintf` evaluates a template-metafunction at compilation time which verifies the number and type of the arguments. `safePrintf` emits a compilation error [14] when at least one of the arguments is not correct. If the evaluation succeeds `safePrintf` calls `printf` with the same arguments `safePrintf` was called with. The template metafunction verifying the arguments has only compilation time overhead, it has zero runtime overhead, the body of `safePrintf` consists of a call to `printf` which is likely to be inlined. At the end of the day using `safePrintf` has zero runtime overhead compared to `printf`. Here is a sample implementation of our `safePrintf`:

```
template <typename FormatString, typename A1, typename A2>
int safePrintf(A1 a1, A2 a2)
{
  BOOST_STATIC_ASSERT((
    CheckArguments<
      FormatString,
      boost::mpl::list<A1, A2>
    >::type::value
  ));
  return
    printf(boost::mpl::c_str<FormatString>::type::value, a1, a2);
}
```

This example works only when exactly two arguments are passed to `safePrintf`. We will generalise it later. We evaluate a metafunction called `CheckArguments` which takes the format string, which is a compilation time string, and a type list containing the types of the arguments passed to `printf`. `CheckArguments` evaluates to a `bool` value: it is `true` when the argument types are valid and it is `false` when they are not. `CheckArguments` parses the format string character by character and verifies that the arguments conform to the format string.

After verifying the validity of the arguments `safePrintf` generates code calling the original `printf` function of the C library. The format string passed to `printf` is automatically generated from the compilation time string argument of `safePrintf`. For example

```
safePrintf< string<'Hell', 'o %s', '!'> >("John");
```

calls `printf` with the following arguments:

```
printf("Hello %s!", "John");
```

Under the hood `CheckArguments` uses a finite state machine [13] to parse the format string. The states of the machine are represented by template metafunctions, the state transitions are done by the C++ compiler during template metafunction evaluation. Template metafunctions are evaluated lazily, thus the C++ compiler instantiates only valid state transitions of the finite state machine. When an argument of `safePrintf` has the wrong type according to the format string, `CheckArguments` stops immediately, skipping further state transitions of the finite state machine. Thus the C++ compiler has a chance to emit the error immediately and continue compilation of the source code. We use a helper function, `CheckArgumentsNonemptyFormatString`, to implement `CheckArguments`:

```
template <typename FormatString, typename Ts>
struct CheckArgumentsNonemptyFormatString :
  boost::mpl::eval_if<
    typename boost::mpl::equal_to<
      typename boost::mpl::front<FormatString>::type,
      boost::mpl::char_<'%'>
    >::type,
    ParseSpecifier<
      typename boost::mpl::pop_front<FormatString>::type,
      Ts
    >,
    CheckArguments<
      typename boost::mpl::pop_front<FormatString>::type,
      Ts
    > > {};

template <typename FormatString, typename Ts>
struct CheckArguments :
  boost::mpl::eval_if<
    typename boost::mpl::empty<FormatString>::type,
    boost::mpl::empty<Ts>,
    CheckArgumentsNonemptyFormatString<FormatString, Ts>
  > {};
```

As one can see the template metafunction `CheckArguments` is just a wrapper for `CheckArgumentsNonemptyFormatString` to handle empty format strings, the

real parsing is done by `CheckArgumentsNonemptyFormatString`. The combination of these metafunctions represent one state of the finite state machine. Every character except `%` transitions back to this state, those characters are not important for us. The `%` character transitions to another state, represented by the `ParseSpecifier` metafunction:

```
template <typename FormatString, typename Ts>
struct ParseSpecifier : boost::mpl::and_<
    IsArgumentValid<
      typename boost::mpl::front<FormatString>::type::value,
      typename boost::mpl::front<Ts>::type>,
    CheckArguments<
      typename boost::mpl::pop_front<FormatString>::type,
      typename boost::mpl::pop_front<Ts>::type > > {};
```

This metafunction verifies the argument specified by the currently parsed placeholder in the format string using `IsArgumentValid`. When it is ok it continues the verification, otherwise it emits an error immediately. The implementation of `IsArgumentValid` is straightforward, it takes a character constant and a type as its arguments and evaluates to a `bool` value. It can be implemented in a declarative way:

```
template <char specifier, typename Ts>
struct IsArgumentValid : boost::mpl::false_ {};

template <>
struct IsArgumentValid<'c', char> : boost::mpl::true_ {};

template <>
struct IsArgumentValid<'d', int> : boost::mpl::true_ {};
// ...
```

Note that only the implementation of a simplified version of `safePrintf` was presented here to demonstrate how our solution works, the implementation of a verification function supporting the whole syntax of `printf` is too long to discuss here.

We have only shown the implementation of a `safePrintf` taking exactly 2 arguments. Other versions can be implemented in a similar way:

```
template <typename FormatString>
int safePrintf();

template <typename FormatString, typename A1>
int safePrintf(A1 a1);

template <typename FormatString, typename A1, typename A2>
int safePrintf(A1 a1, A2 a2);
```

```
template <typename FormatString,
          typename A1,
          typename A2,
          typename A3>
int safePrintf(A1 a1, A2 a2, A3 a3);
// ...
```

These functions can be automatically generated using the Boost precompiler library [31]. As it is the case with other Boost libraries, the number of `printf` functions generated can be specified by a macro evalutaing to an integer value. Thus, users of the library can increase it according to their needs. We do not present here how we generate these functions, it can be done using `BOOST_PP_REPEAT` provided by the Boost precompiler library.

This solution combines the simple usage and small run-time overhead of `printf` with the type-safety of C++ using compilation time strings. Stroustrup wrote a type-safe `printf` using variadic template functions [10,28] which are part of the upcoming standard, C++0x [21]. His implementation uses run-time format strings and transforms `printf` calls to writing to C++ streams at runtime. For example the code

```
printf("Hello %s!", "John");
```

using his type-safe `printf` does

```
std::cout << 'H' << 'e' << 'l' << 'l' << 'o'
          << ' ' << "John" << '!';
```

at runtime. This solution prints the format string character by character which makes it extremely slow. The author's intention was to demonstrate the use of variadic templates, but it can be further optimized in the following way:

```
std::cout << "Hello " << "John" << "!";
```

We have measured the speed of normal `printf`, used by our implementation, and both of the above. We measured the speed of the following call:

```
printf("Test %d stuff\n", i);
```

and its `std::cout` equivalents. We printed the text 100 000 times and measured the speed using the `time` command on a Linux console. The average time it took can be seen in Table 1. `printf`, which is used by our type-safe implementation, is almost four times faster than the example on [28] and more than two times faster than the optimized version of that example.

We measured the performance of the C style `printf` function and the C++ style `std::cout` stream with several kinds of input from the simple ones to more compound samples. To do this measurement we used the profiler module of Valgrind [35] dynamic analysis tool called Callgrind. Table 2 shows the results.

Table 1. Elapsed time

Method used	Time
std::cout for each character	0,573 s
normal std::cout	0,321 s
printf	0,152 s

Table 2. Instructions fetched

Pattern	printf	cout
"hello"	326	363
"hello%s", "world"	603	722
"hello%s%d", "world", 1	942	1217
"hello%s%d%c", "world", 1, 'a'	1149	1500
"hello%s%d%c\n", "world", 1, 'a'	1395	2148

In the first column we present the printed pattern. The second column shows the instructions needed to print it using `printf`, and the third one shows the same using `cout`.

As we can see from the table, `cout` is slower than `printf`. When the printed text is simple, the difference is slight, but it is growing as the text becomes more and more complex.

Another difference between Stroustrup's type-safe `printf` and ours is the way they validate the types of the arguments. Stroustrup's solution ignores the type specified in the format string, it displays every argument supporting the streaming operator regardless of its type. For example, it accepts the following incorrect usage of `printf`

```
printf("Incorrect: %d", "this argument should be an integer");
```

while our solution emits an error at compilation time. On the other hand, our solution can only deal with types the C `printf` can handle, while Stroustrup's solution can deal with any type which supports the streaming operator.

A drawback of Stroustrup's solution is that it does not detect when the arguments of `printf` are shifted or are in the wrong order and displays them incorrectly. For example Stroustrup's `printf` accepts

```
printf("Name: %s\nAge: %d\n", "27", "John");
```

and displays

```
Name: 27
Age: John
```

while our solution emits a compilation error.

Stroustrup's solution throws an exception at runtime when the number of arguments passed to `printf` is incorrect, which can lead to hidden bugs due to incomplete testing. Our solution emits compilation errors in such cases to help detecting these bugs.

6 Related Work

Modern programming languages with object-oriented features and operator overloading are able to create classes with an interface close to high level matematical notations. For instance, arrays, matrices, linear algebraic operations are typical examples. However, as Veldhuizen noted, the code generated by such libraries tends to be inefficient [27]. As an example, he measured array objects using operator overloading in C++ were 3-20 times slower than the corresponding low level implementation. This is not because of poor design on the part of library developers, but because object-oriented languages force inefficient implementation techniques: dynamic memory allocations, high number of object copying, etc. These performance problems are commonly called as *abstraction penalty*.

Attempts to implement smarter optimizers were largely unsuccessful, mainly because of the lack of semantical information. Efforts to describe semantics of a type is still in experimental phase without too much result [11]. On the other hand, a more promising approach is to write an *active library*. Active libraries [24] act dynamically, makes decisions at compilation time based on the calling context, choose algorithms, and optimize code. These libraries are not passive collections of functions or objects, like traditional libraries, but take an active role in generating code. Active libraries provide higher abstractions and can optimize those abstractions themselves. In C++ active libraries are implemented with the help of template metaprogramming techniques [13].

An other possible optimization technique is *partial evaluation* [8,12]. Partial evaluators regard a program's computation as containing two subsets: static computations which are performed at compile time, and dynamic computations performed at run time. A partial evaluator executes the static optimizations and produces a specialized *residual* program. To determine which portions of a program can be evaluated, a partial evaluator may perform binding time analysis to separate static and dynamic data and language constructs. Sometimes we call such a language as two-level language. C++ template resemble a two-level language, as function templates take both statically bound template parameters and dynamically bound function arguments [27].

There are third party libraries to apply string-related compilation time operations in some special areas of programming. The `boost::spirit` library is an object oriented recursive descent parser framework [33]. EBNF grammars can be written with C++ syntax and these grammars can be inlined in the C++ source code. Since the implementation of `spirit` uses template metaprogramming techniques, the parser of the EBNF grammar is generated by the C++ compiler. The `boost::wave` C++ preprocessor library [37] uses the `spirit` parser construction library to implement a C++ lexer with ISO/ANSI Standards conformant preprocessing capabilities. Wave provides an iterator interface which gives access to the currently preprocessed token of the input stream. These preprocessed tokens are generated on-the-fly while iterating over the preprocessor iterator sequence. The `boost::xpressive` is a regular expression template library [38] dealing with static regular expressions. This library can perform syntax checking and generate optimized code for static regexes.

Stroustrup demonstrates how a type-safe `printf` can be built using the features of the upcoming C++ standard [21]. The differences between this and our solution are explained in chapter 5.

Since the style of metaprograms is unusual and difficult, it requires high programming skills to write. Maintenance of template metaprograms are much more harder. Besides, it is sorely difficult to find errors in template metaprograms. Porkoláb et al. provided a metaprogram debugger tool[17] in order to help finding bugs.

7 Summary and Future Work

Strings, one of the most commonly used data types in programming, had only weak support for C++ template metaprograms. In this paper we emphasize the importance of string manipulation at compile time. We have developed Metastring library based on `boost::mpl::string` and extended its compile time functionality with the usual operations of run time string libraries. We presented the implementational details of our Metastring library and discussed syntactic simplifications to reduce the syntactical overhead. To illustrate the importance of the metastring, we investigated two application areas in details.

When either the text or the pattern are known at compilation time, pattern matching algorithms can be significally improved. We dealt with two pattern matching algorithms: Boyer-Moore and Knuth-Morris-Pratt to demonstrate the power of metastrings. Our future work is to create a more sophisticated method – which takes more pattern matching algorithms into account – to find the best pattern matching solution.

As the other motivating application, we have created a C++ wrapper for `printf` function taking the format string as a compilation time string argument and validating the type of the runtime arguments based on that string. Validation happens at compilation time, therefore our solution has zero run time overhead but ensures type-safety. We have compared our type-safe `printf` solution to the one on Stroustrup's website and found that our solution provides stricter type-safety and runs at least two times faster. Our future plan is to introduce the `%a` specifier – which means *any* – to force the compiler to deduce the argument's type. Stroustrup's solution behaves similarly.

References

1. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, Reading (2001)
2. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley, Reading (2004)
3. Boyer, R.S., Moore, J.S.: A Fast String Searching Algorithm. Communication of the ACM 20, 762–772 (1977)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Cambridge (2001)

5. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools and Applications. Addison-Wesley, Reading (2000)
6. Czarnecki, K., Eisenecker, U.W., Glück, R., Vandevoorde, D., Veldhuizen, T.L.: Generative Programming and Active Libraries. Springer, Heidelberg (2000)
7. Devadithya, T., Chiu, K., Lu, W.: C++ Reflection for High Performance Problem Solving Environments. In: Proceedings of the 2007 Spring Simulation Multiconference, vol. 2, pp. 435–440 (2007)
8. Futamura, Y.: Partial Evaluation of Computation Process – An approach to a Compiler-Compiler. Systems, Computers, Controls 2(5), 45–50; Reprinted in Higher-Order and Symbolic Computation 12(4), 381–391 (1999), with a foreword. Kluwer Academic Publishers (2000)
9. Gil, J.(Y.), Lenz, K.: Simple and safe SQL queries with c++ templates. In: Proc. of Generative And Component Engineering 2007, The ACM Digital Library, pp. 13–24 (2007)
10. Gregor, D., Järvi, J., Powell, G.: Variadic Templates (Revision 3). Number N2080=06-0150, ANSI/ISO C++ Standard Committee (October 2006)
11. Gregor, D., Järvi, J., Kulkarni, M., Lumsdaine, A., Musser, D., Schupp, S.: Generic programming and high-performance libraries. International Journal of Parallel Programing 33(2), 145–164 (2005)
12. Jones, N.D.: An introduction to partial evaluation. ACM Computing Surveys 28(3), 480–503 (1996)
13. Juhász, Z., Sipos, Á., Porkoláb, Z.: Implementation of a Finite State Machine with Active Libraries in C++. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 474–488. Springer, Heidelberg (2008) ISBN 978-3-540-88642-6
14. Karlsson, B.: Beyond the C++ Standard Library, An Introduction to Boost. Addison-Wesley, Reading (2005)
15. Knuth, D.E., Morris Jr., J.H., Pratt, V.R.: Fast Pattern Matching in Strings. SIAM J. Comput. 6(2), 323–350 (1977)
16. Meyers, S.: Effective STL. Addison-Wesley, Reading (2001)
17. Porkoláb, Z., Mihalicza, J., Sipos, Á.: Debugging C++ Template Metaprograms. In: Proc. of Generative Programming and Component Engineering (GPCE 2006), The ACM Digital Library, pp. 255–264 (2006)
18. ANSI/ISO C++ Committee. Programming Languages – C++. ISO/IEC 14882:1998(E). American National Standards Institute (1998)
19. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: First Workshop on C++ Template Metaprogramming (October 2000)
20. Sipos, Á., Porkoláb, Z., Pataki, N., Zsók, V.: Meta < Fun > - Towards a Functional-Style Interface for C++ Template Metaprograms. In: Proceedings of 19th International Symposium of Implementation and Application of Functional Languages (IFL 2007), pp. 489–502 (2007)
21. Stroustrup, B.: Evolving a language in and for the real world: C++ 1991-2006. ACM HOPL-III (June 2007)
22. Stroustrup, B.: The C++ Programming Language Special Edition. Addison- Wesley, Reading (2000)
23. Unruh, E.: Prime number computation. ANSI X3J16-94-0075/ISO WG21-462
24. Veldhuizen, T.L., Gannon, D.: Active libraries: Rethinking the roles of compilers and libraries. In: Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientic and Engineering Computing (OO 1998), pp. 21–23. SIAM Press, Philadelphia (1998)

25. Veldhuizen, T.L.: Expression Templates. C++ Report 7(5), 26–31 (1995)
26. Veldhuizen, T.L.: Using C++ Template Metaprograms. C++ Report 7(4), 36–43 (1995)
27. Veldhuizen, T.L.: C++ Templates as Partial Evaluation. In: Danvy, O. (ed.): Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, January 22-23. Technical report BRICS-NS-99-1, University of Aarhus, pp. 13–18 (1999)
28. http://www.research.att.com/~bs/C++0xFAQ.html
29. Boost Concept checking, http://www.boost.org/libs/concept_check/concept_check.htm
30. Boost Metaprogramming library, http://www.boost.org/libs/mpl/doc/index.html
31. The boost preprocessor metaprogramming library, http://www.boost.org/doc/libs/1_41_0/libs/preprocessor/doc/index.html
32. The boost proto library, http://www.boost.org/doc/libs/1_37_0/doc/html/proto.html
33. http://spirit.sourceforge.net
34. http://www.boost.org/doc/libs/1_42_0/libs/mpl/doc/refmanual/string.html
35. http://valgrind.org
36. http://www.boost.org/doc/libs/1_40_0/libs/mpl/doc/refmanual/vector-c.html
37. http://www.boost.org/doc/libs/1_40_0/libs/wave/index.html
38. The boost xpressive regular library, http://www.boost.org/doc/libs/1_40_0/doc/html/xpressive.htm

Language Convergence Infrastructure

Vadim Zaytsev

Software Languages Team, Universität Koblenz-Landau, Germany
zaytsev.vadim@gmail.com

Abstract. The process of grammar convergence involves grammar extraction and transformation for structural equivalence and contains a range of technical challenges. These need to be addressed in order for the method to deliver useful results. The paper describes a DSL and the infrastructure behind it that automates the convergence process, hides negligible back-end details, aids development/debugging and enables application of grammar convergence technology to large scale projects. The necessity of having a strong framework is explained by listing case studies. Domain elements such as extractors and transformation operators are described to illustrate the issues that were successfully addressed.

1 Introduction

The method of grammar convergence has been presented in [15] and elaborated in a large case study [16], with a journal version being in print. The basic idea behind it is to extract grammars from available grammar artefacts, transform

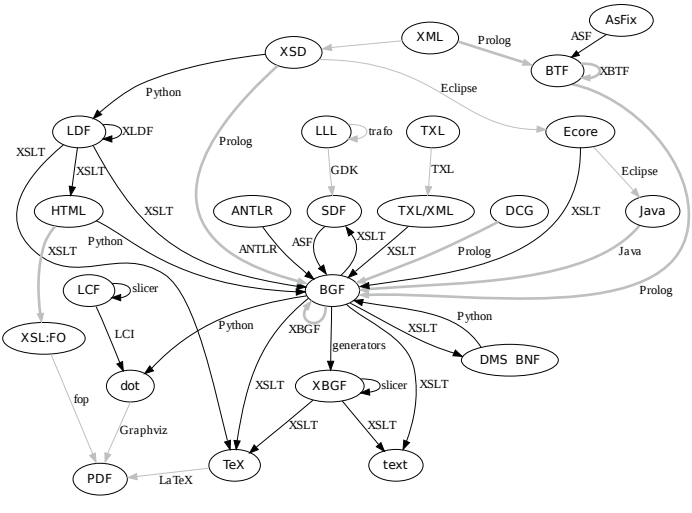


Fig. 1. The megamodel of SLPS: every vertex is a language, every arc is a language transformation. Thin grey lines denote tools present prior to this research: e.g., GDK [13] or TXL [3]. Thick grey edges are for co-authored transformations.

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 481–497, 2011.

them until they become identical, and draw conclusions from the properties of the transformation chain: its length, the type of steps it consisted of, the correspondence with the properties expected a priori from documentation, etc. Grammar convergence can be used among other ways to establish an agreement between a hand-crafted object model for a specific domain and an XML Schema for standard serialisation of the same domain; to prove that various grammarware such as parsers, code analysers and reverse engineering tools agree on the language; to synchronise the language definition in the manual with the reference implementation; to aid in disciplined grammar adaptation.

In this paper we will use the terms "grammar convergence" and "language convergence" almost interchangeably. In fact, language convergence is a broader term that includes convergence of not only the syntax, but also parse trees, documentation, possibly even semantics. We focus on dealing with grammars here, but the reader interested in consistency management for language specifications can imagine additional automated steps like extracting a grammar from the language document before the transformation and inserting it back afterwards [12,14].

Language convergence was developed and implemented as a part of an open source project called SLPS, or Software Language Processing Suite[1]. It comprises several stand-alone scripts targeting comparison, transformation, benchmarking, validation, extraction, pretty-printing. Most of those scripts were written in Python, Prolog, Shell and XSLT. Grammar convergence is a complicated process that can only be automated partially and therefore requires expert knowledge to be used successfully. In order to simplify the work of a grammar engineer, a specific technical infrastructure is needed with a solid transformation operators suite, steadily defined internal notations and a powerful tool support for every stage. This paper presents such a framework and explains both engineering and scientific design choices behind it.

Figure 1 presents a "megamodel" [2] of SLPS. Every arc from this graph is a language transformation tool or a sequence of pipelined tools. Many of the new DSLs developed for this infrastructure are in fact XML: BGF, XBGF, BTF, XBTF, LDF, XLDF, LCF—just an engineering decision that let them profit fully from XMLware facilities like validation against schemata and transformation with pattern matching. (These advantages are not unique for XML, of course). Others are mostly well-known languages that existed prior to this research: ANTLR [18], SDF [9], LLL [13], XSD [5], etc.

The left hand side of the megamodel is mostly dedicated to language documentation-related components: LDF is a Language Document Format [23], an extension of grammar notation that covers most commonly encountered elements of language manuals and specifications. The central part contains the grammar notation itself: the BGF node has a big fan-in since every incoming arc represents a grammar extraction tool (see §4.1). The only outgoing arcs are the main presentation forms: pure text, marked up LaTeX and a graph form, plus transformation generators (see §5.4) and integration tools (see §6.3).

[1] Software Language Processing Suite: `http://slps.sf.net`

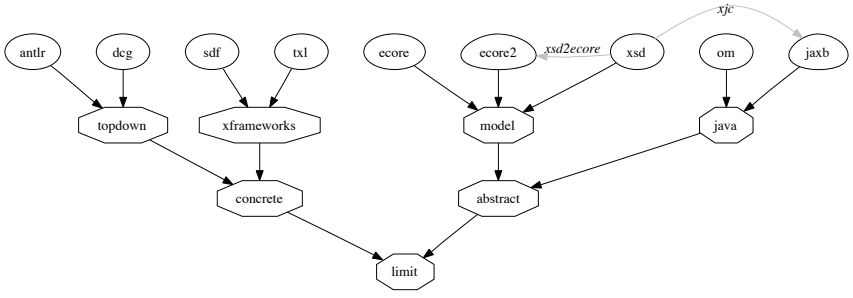


Fig. 2. The overall convergence graph for the Factorial Language. The grey arrows show grammar relations that are expressed in LCF but not performed directly by the convergence infrastructure (the reason is that, for example, generating Ecore from XML Schema cannot be done from command line and must be performed via Eclipse IDE).

The inherent complexity of the domain and the methodology led to the development of what we call LCI, or Language Convergence Infrastructure. It is the central point of SLPS, it provides full technical support to its functionalities, operating on a DSL called LCF (LCI Configuration Format) in which the input configuration must be expressed. The DSL details are also provided in this paper.

§2 motivates the need for language convergence by giving three example scenarios of its application. §3 starts describing the domain by explaining the DSL, while §4, §5 and §6 address the notions linked to sources, transformations and targets correspondingly.

2 Motivation

In this section three distinct applications of grammar convergence are briefly presented together with the results acquired from them.

2.1 Same Language, Multiple Implementations: Factorial Language

A trivial functional programming language was defined in [15] to test out the method of grammar convergence, we called it Factorial Language. We modelled the common scenario of one language having several independently developed grammars by writing or generating nine grammar artefacts within various frameworks, as seen on Figure 2:

antlr. A parser description in the input language language of ANTLR [18]. Semantic actions (in Java) are intertwined with EBNF-like productions.

dcg. A logic program written in the style of definite clause grammars.

sdf. A concrete syntax definition in the notation of SDF (Syntax Definition Formalism [9]), a parser description targeted for SGLR parsing.

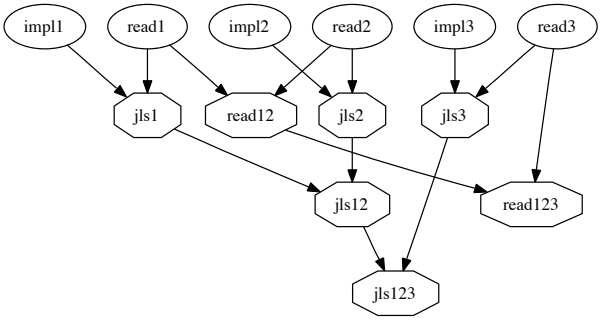


Fig. 3. Binary convergence tree for the JLS grammars—or rather two trees with shared leaves. As usual, the nodes on the top (the leaves) are grammars extracted directly from the JLS. All other nodes are derived by transformation chains denoted as arcs. We use a (cascaded) binary tree here: i.e., each non-leaf node is derived from two grammars.

txl. Another transformational framework that allows for agile development of tools based on language descriptions [3].

ecore. An Ecore model, created manually in Eclipse and represented in XMI [17].

ecore2. An alternative Ecore model, generated automatically by Eclipse, given the XML Schema of the domain.

xsd. An XML schema [5] for the abstract syntax of FL. In fact, this is the schema that served as the input for generating both the object model of the *jaxb* source and the Ecore model of the *ecore2* source.

om. A hand-crafted object model (Java classes) for the abstract syntax of FL. It is used by a Java-based implementation of an FL interpreter.

jaxb. Also an object model, but generated by the JAXB data binding technology [10] from the XML schema for FL.

2.2 Language Evolution: Java Language Specification

In [16] we describe a completed effort to recover the relationships between all the grammars that occur in the different versions of the Java Language Specification (JLS). The case study concerns the 3 different versions of the JLS [6,7,8] where each of the 3 versions contains 2 grammars: one grammar is optimised for readability (i.e., *read1–read3* on Figure 3), and another one is intended to serve as a basis for implementation (i.e., *impl1–impl3* on Figure 3). The JLS is critical to the Java platform — it is a foundation for compilers, code generators, pretty-printers, IDEs, code analysis and manipulation tools and other grammarware for the Java language. One would expect that the different grammars per version are essentially equivalent in terms of the generated language. For implementability reasons one grammar may be more liberal than the other. One would also expect that the grammars for the different versions engage in an inclusion ordering (again, in terms of the generated languages) because of the backwards-compatible evolution of the Java language.

The case study comprised around 17000 transformation steps and has shown that the expected relationships of (liberal) equivalence and inclusion ordering are significantly violated by the JLS grammars. Thus, grammar convergence can be used as a form of consistency management for the JLS in particular, and language specifications in general.

2.3 BNF-Like Grammar Format

The abstract syntax of BGF, which is the internal representation for grammars in our infrastructure, is defined by the corresponding XML Schema. There is also a pretty-printer that helps to present BGF grammars for debugging and publishing purposes (let us call this presentation notation "BNF"). This pretty-printer is grammarware, the concrete syntax of its output can be specified by a grammar. How does this grammar relate to the XML Schema of BGF? We applied grammar convergence method to these two grammars: the hand-crafted one for the concrete syntax and the one derived from the XSD for the abstract syntax. (We had to be satisfied with a manually engineered grammar since grammar inference from an XSL transformation sheet is far from trivial and perhaps even undecidable).

The convergence graph is trivial and thus not shown here. The transformation scripts are also considerably simple for this case study, which allowed us to examine them in detail. The conclusion was that: **BGF allows for empty grammars, BNF does not** (as expected, since BNF is used for presentation); **BNF contains indentation rules and abstract syntax, BGF does not** (as expected due to the abstract nature of BGF); **BGF includes root elements, BNF does not** (as expected, since EBNF dialects never specify starting symbols).

This case study shows that grammar convergence can also be used for validating implicit assumptions within a grammar engineering infrastructure. The cost of such use is low (the case study took no more than an hour), but it brings more discipline to the grammar engineering process. The additional confidence comes from the guarantee that there are no other differences besides those included in our list.

3 Grammar Convergence Domain Overview

Grammar convergence is a method of establishing relationships between language grammars by extracting them and transforming towards equivalence. Thus, we distinguish three core domain elements: the **source grammars** that are obtained from available grammar artefacts; the **target grammars** that are the common denominators of the source grammars; and the **transformation chains** that bind them together and represent their relationships, as shown on Figure 4. The methodology has been presented in [15] and elaborated in a large case study [16].

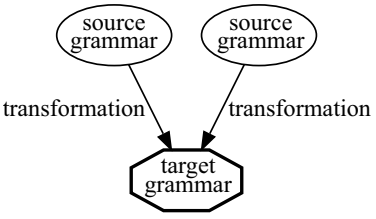


Fig. 4. The abstract view on the grammar convergence process

LCF is a configurational domain specific language that is used by the LCI. Since it encapsulates all crucial domain concepts, we will examine its grammar and explain them while doing so. The grammar is presented in an EBNF dialect specific for SLPS: beside the usual notation it has selectors. By writing `a::b` we refer to a nonterminal `b` but label its particular occurrence as `a`. There are also four built-in symbols: ***string*** for any string, ***xstring*** for a macro expanded string, ***id*** for a unique identifier denoting an entity such as a grammar or a tool and ***refid*** for a reference to such an identifier.

```
scenario:
       shortcut* tools source+ target+ testset*
shortcut:
       name::id expansion::xstring
```

Each convergence scenario contains shortcuts, tools, sources, targets and test sets. **Shortcuts** are macro definitions used mostly for maintainability purposes: for example, it is possible with them to define the path to the main working directory once and refer to it in all necessary places. Shortcuts can be defined based on other shortcuts.

```
tools:
       transformer::tool comparator::tool validator::tool? generator*
tool:
       grammar::xstring tree::xstring?
```

Two **tools** are crucial for grammar convergence and must always be defined: the transformer and the comparator. The **transformer** takes a BGF grammar and an XBGF script and applies the latter to the former, resulting in a transformed BGF grammar (or an error return code, which is handled by the LCI). §5 will address this tool in detail. The **comparator** takes two BGF grammars and returns the verdict on their equivalence. Since the premise of grammar convergence method was to document grammar relationships, the comparator is not expected to do any sophisticated matching besides applying basic algebraic laws. A **validator** tool is optional and can check each one of the many XML files generated in the convergence process for well-formedness and conformance to a schema. Both tools will be described in §6. Each of these tools can consist of a pair of references to external programs: one program that operates on a grammar level and one on a parse tree level. The latter part is optional, but if it is absent, no coupled transformations can take place.

```
generator:
       name::id command::xstring
```

A transformation **generator** is a named tool that takes a BGF grammar as an input and produces an XBGF script applicable to that grammar and containing transformations of a certain nature, see §5.

```
testset:
       name::id command::xstring
```

A **test set** is also used for coupled transformations and for more thorough validation: each test case is tried with a corresponding parser and is co-transformed. This subdomain will not be addressed in this paper since it is a separate big research area and still work in progress for us.

4 Convergence Sources

```
source:
       name::id derived? source-grammar source-tree? test-set::refid*
derived:
       from::refid using::string
source-grammar:
       extraction::xstring parsing::xstring? evaluation::xstring?
source-tree:
       extraction::xstring evaluation::xstring?
```

A convergence **source** is defined at least by a name and the command that will be executed for its **extraction**. Possible additional properties include for a **derived** source its previously known (but invisible for LCI otherwise) relation to another source, which allows LCI to draw grey links in Figure 2. A grammar-based **parser** and **evaluator** detailed in the next subsections, can also be specified. The **extractor** and **evaluator** for the tree level are also optional (the corresponding parser does not make sense since a parse tree is stored in a BTF which is either correct by definition or filtered out by a validator). **Test sets** compatible with this source can also be listed here to be used later to find bugs in the source grammar.

One of the crucial parts of our infrastructure is the format for storing grammars. Instead of trying to model all possible peculiar or even idiosyncratic details deployed within grammar artefacts in various frameworks: semantic actions, lexical syntax descriptions, precedence declarations, classes/interfaces or elements/attributes dichotomy, etc—we opted for sacrificing them and storing only the crucial core grammar knowledge. In fact, by abstracting from these details at the extraction stage, we get an XML-based dialect of EBNF.

4.1 Extractors

Extraction happens only once per source even if the source is used more than once. When it succeeds, LCI stores the extracted grammar in order to fall back to the old snapshot if it ever goes wrong in one of the future runs. The extracted grammar is also subject to validation, in case the validator is specified.

An extractor is simply a software component that processes a software artefact and produces a BGF grammar. In the simplest case, extraction boils down to a straightforward mapping defined by a single pass over the input. Extractors are

Table 1. The mapping between the XML output of the TXL parser and BGF

TXL	BGF
`program`	`bgf:grammar`
`defineStatement`	`bgf:production`
`repeat barLiteralsAndTypes`	`bgf:expression/choice`
`repeat literalOrType`	`bgf:expression/sequence` if length>1
`literalOrType/type/typeSpec` (depending on `opt typeRepeater/typeRepeater`)	`bgf:expression/plus/bgf:expression` or `bgf:expression/star/bgf:expression` or `bgf:expression/optional/bgf:expression` or `bgf:expression`
`literalOrType/literal`	`bgf:expression/terminal`
`typeid/id`	`nonterminal`

typically implemented within the computational framework of the kind of source, or in its affinity: e.g., in Prolog for DCG, in ASF+SDF for SDF, in ANTLR for ANTLR. Several examples follow.

TXL to BGF mapping. TXL [3] distribution contains a TXL grammar for TXL grammars. By using that, we can parse any correct TXL grammar and serialise the resulting abstract syntax tree in the XML form. After that the mapping becomes trivial and is easily implemented in the form of XSLT templates that match TXL tags and generate BGF tags with the equivalent internal details, as shown on Table 1.

SDF to BGF mapping. The Meta-Environment [11] contains both SDF definition for SDF definitions and the transformational facitilies needed for mapping. After specifying the mapping in ASF in the form of traversal functions and rewriting rules, this sequence of actions is required for extraction:

- ◇ `pack-sdf` for combining all extractor modules into one definition
- ◇ `sdf2table` for making a parse table out of that definition
- ◇ `eqs-dump` for compiling ASF formulæ
- ◇ `sglr` for parsing the SDF source grammar with the table
- ◇ `asfe` for rewriting the resulting parse tree
- ◇ `unparsePT` for serialising the transformed parse tree into the file

These tools are tied together by appropriate makefiles and shell scripts. The first three steps are performed once and need to be redone only if the extractor itself changes; the last three steps are executed per extracted grammar.

HTML to BGF recovery. A JLS document is basically a structured text document with embedded grammar sections. In fact, the more readable grammar is developed throughout the document where the whole more implementable grammar is given at once in the last section.

Table 2. Irregularities resolved by grammar extraction given the HTML source

	impl1	impl2	impl3	read1	read2	read3	Total
Arbitrary lexical decisions	2	109	60	1	90	161	423
Well-formedness violations	5	0	7	4	11	4	31
Indentation violations	1	2	7	1	4	8	23
Recovery rules	3	12	18	2	59	47	141
◦ Match parentheses	0	3	6	0	0	0	9
◦ Metasymbol to terminal	0	1	7	0	27	7	42
◦ Merge adjacent symbols	1	0	0	1	1	0	3
◦ Split compound symbol	0	1	1	0	3	8	13
◦ Nonterminal to terminal	0	7	3	0	8	11	29
◦ Terminal to nonterminal	1	0	1	1	17	13	33
◦ Recover optionality	1	0	0	0	3	8	12
Purge duplicate definitions	0	0	0	16	17	18	51
Total	11	123	92	24	181	238	669

The JLS is available electronically in HTML and PDF format. Neither of these formats was designed with convenient access to the grammars in mind. We have opted for the HTML format here. The grammar format slightly varies across the different JLS grammars and versions; we had to reverse engineer formatting rules from different documents and sections — in particular from [6,7,8, §2.4] and [7,8, §18].

In order to deal with irregularities of the input format, such as liberal use of markup tags, misleading indentation, duplicate definitions as well as numerous smaller issues, we needed to design and implement a non-classic parser to extract and analyse the grammar segments of the documents and to perform a recovery. About 700 fixes were performed that way, as can be seen from Table 2.

We face a few syntax errors with regard to the syntax of the grammar notation. We also face a number of "obvious" semantic errors in the sense of the language generated by the grammar. We call them obvious errors because they can be spotted by simple, generic grammar analyses that involve only very little Java knowledge, if any. We have opted for an error-recovery approach that relies on a uniform, rule-based mechanism that performs transformations on each sequence of tokens that corresponds to an alternative.

The rules are implemented in Python by regular expression matching. They are applied until they are no longer applicable. Examples of them include matching up missing parentheses by deriving their absence from the context, converting improperly positioned metasymbols to terminals and removing duplicate definitions. The complete list is given with details and examples in the journal version of [16].

4.2 Parsers and Evaluators

A **parser** is one of the most commonly available grammar artefacts: it is a syntactic analyser that can tell whether some input sequence matches the given

grammar. If such a tool is indeed present, it can be referenced in LCF as well and will be used for testing purposes. A compatible test set must also be provided separately.

It is possible to implement all transformation operators to be applicable not only to languages (grammars), but also to instances (parse trees). If this is done and the corresponding tree extractors and parsers are provided in LCF, then LCI is not limited to converging grammars only. For every source that has a test set attached, for every test case in that set, LCI performs coupled extraction, transformation and comparison.

Additionally, **evaluators** can be provided that can execute test cases and compare return values with expected ones (for simplicity our prototype works with integers). Test sets must be present in a unified format for LCI to figure out applicable actions. Test cases will also be validated if the validation tool is specified. The evaluators play a similar role, but their return value is not an error code, but rather the result of evaluating the given expression. The difference between an evaluator listed in the grammar properties and an evaluator given in the instance properties is that the input of the former is a correct program in the original format and the latter takes a parse tree of an instance, presented in BTF (BGF Tree Format).

5 Grammar Transformation

We have developed a suite of sophisticated grammar transformation operators that can be parametrised appropriately and called from a script. The resulting language is called XBGF (X stands for transformation), and is processed by the transformer. Some XBGF commands have been presented in [15,16], we give several examples here as well; the complete language manual is available as [22].

5.1 Unfolding

There are several folding and unfolding transformation operators in XBGF, of which the simplest one is just called **unfold**. It searches the scope for all the instances of the given nonterminal usage and replaces such occurrences with the defining expression of that nonterminal. By default the scope of the transformation is the full grammar, but it can be limited to all the definitions of one nonterminal or to one labelled production. Regardless of the specified scope, unfolding is not applied to the definition of the argument nonterminal.

The definition that is being unfolded is assumed to consist of one single production. When only one of several existing productions is used for unfolding, such a transformation makes the language (as a set of strings generated by the context-free grammar) smaller. The corresponding XBGF command is called **downgrade**. Other refactoring variants of **unfold** operator include **inline** that unfolds the definition and purges it from the grammar, and **unchain** which removes chain productions (`a: b; b: ...;` with no other use for `b`).

5.2 Massaging

The **massage** operator is used to rewrite the grammar by local transformations such that the language generated by the grammar (or the denotation according to any other semantics for that matter) is preserved. There are two expression arguments: one to be matched, and another one that replaces the matched expression. One of them must be in a "massage relation" to the other. The scope of the transformation can be limited to one labelled production or to all productions for a specific nonterminal symbol.

The massage-equality relation is defined by these algebraic laws:

$$
\begin{array}{lll}
x? = (x;\varepsilon) & (x?)? = x? & (x, x^\star) = x^+ \\
x? = (x?;\varepsilon) & (x?)^+ = x^\star & (x^\star, x) = x^+ \\
x^\star = (x^+;\varepsilon) & (x?)^\star = x^\star & (x?, x^\star) = x^\star \\
x^\star = (x^\star;\varepsilon) & (x^+)? = x^\star & (x^\star, x?) = x^\star \\
x? = (x?;x) & (x^+)^+ = x^+ & (x^+, x^\star) = x^+ \\
x^+ = (x^+;x) & (x^+)^\star = x^\star & (x^\star, x^+) = x^+ \\
x^\star = (x^\star;x) & (x^\star)? = x^\star & (x^+, x?) = x^+ \\
x^\star = (x?;x^+) & (x^\star)^+ = x^\star & (x?, x^+) = x^+ \\
x^\star = (x?;x^\star) & (x^\star)^\star = x^\star & (x^\star, x^\star) = x^\star \\
x^\star = (x^+;x^\star) & x = (s_1 :: x; s_2 :: x) &
\end{array}
$$

The selectors are needed in the bottom right formula because a choice between two unnamed x will always be normalized as x, as explained in §6.1.

5.3 Projection and Injection

A good example of the transformation operators that do not preserve semantics of a language will be **inject** and **project**. Projection means removing components of a sequential composition, injection means adding them. The operators take one production as a parameter with additional or unnecessary components marked in a special way. For projection the transformation engine checks that the complete production exists in the grammar and replaces it with the new production with fewer components, injection works similarly, but the other way around. If the projected part is nillable, i.e. it can evaluate to ε, the operator is always semantic-decreasing and is called **disappear**. If the projected part corresponds to the concrete syntax, i.e. contains only terminal symbols, the operator preserves abstract semantics and is called **abstractize**.

5.4 Transformation Generators

Grammar convergence research has started with an objective to use programmable grammar transformations to surface the relationships between grammars extracted from sources of different nature. Hence, we mostly aimed to provide a comprehensive transformation suite, a convergence strategy and an infrastructure support.

However, at some point we found it easier to generate the scripts to resolve specific mismatches rather than to program them manually. A full-scale research on this topic remains future work, yet below we present the results obtained so far and the considerations that can serve as foundation for the next research steps.

Consider an example of converging concrete and abstract syntax definitions. This situation requires a transformation that removes all details that are specific to concrete syntax definitions, i.e., first and foremost strips all the terminals away from the grammar. Given the grammar, it is always possible to generate a sequence of transformations that will remove all the terminal symbols. It will take every production in the grammar, search for the terminals in it and if found, produce a corresponding call to **abstractize** (the name refers to going from concrete syntax to abstract syntax). For instance, given the production:

```
[ifThenElse] expr:
       "if" expr "then" expr "else" expr
```

The following transformation will be generated (the angle brackets denote parts that will be projected away):

```
abstractize(
 [ifThenElse] expr:
       ⟨"if"⟩ expr ⟨"then"⟩ expr ⟨"else"⟩ expr
);
```

Other generators we used in the FL case study were meant for removing all selectors from the grammar (works quite similar to removing terminals), for disciplined renamings (e.g., aligning all names to be lower-case) and for automated setting of the root nonterminals by evaluating them to be equal to top nonterminals of the grammar.

Eliminating all unused nonterminals can also be a valuable generator in some cases. For us it was not particularly practical since we wanted to look into each nominal difference (which unused terminal is a subtype of) in order to better align the grammars.

A more aggressive transformation generator example can be the one that **inline**s or **unchain**s all nonterminals that are used only once in the grammar. This can become a powerful tool when converging two slightly different grammars and thus can be considered a form of aggressive normalisation. We did not work out such an application scenario for grammar convergence so far.

Deyaccification [12,20], a well-studied mapping between recursion-based and iteration-based nonterminal definitions, can also be performed in an automated fashion. In general, all grammar transformations that have a precondition enabling their execution, can be generated—we only need to try to apply them everywhere and treat failed preconditions as identical transformations.

On various occasions we also talk about "vertical" and "horizontal" productions. The former means having separate productions for one nonterminal, as in:

```
x:
       foo
x:
       bar
```

The latter (horizontal) means having one production with a top choice, as in:

```
x:
        foo
        bar
```

There are also singleton productions that are neither horizontal nor vertical (as in just "`x: foo`"), and productions that can be made horizontal by distribution (as in "`x: foo | bar`"). According to this classification and to the need of grammar engineers, it is possible to define a range of generators of different aggressiveness levels that would search for horizontal productions and apply **vertical** to them; or search for vertical productions and apply **horizontal** to them; or search for potential horizontal productions and apply **distribute** and **vertical** to them; etc.

It is important to note here that even though complete investigation of the possible generators and their implementation remain future work, this alone will not be enough to replace human expertise. Semi-automation will only be shifted from "choose which transformation to apply" to "choose which generator to apply". A strongly validated strategy for automating the choice is needed, which is not easy to develop, even if possible.

6 Convergence Targets

A target needs a name and one or more branches it consists of:

```
target:
        name::id branch+
branch:
        input::refid preparation::phase? nominal-matching::phase? structural-matching::phase?
                (extension::phase | correction::phase | relaxation::phase)*
```

Each branch is defined as an input node and optionally some phases. The input can refer to a source or to another target, which is then called an **intermediate target**. Phases of convergence have been related to the strategy advised by [16], the notion is used to separate preliminary nominal matching scripts from language-preserving refactorings doing structural matching and from unsafe steps like relaxation, correction or extension. In case of no phases specified, the input grammar is propagated to the output of the branch.

```
phase:
        step::(perform-transformation::string | automated-transformation)+
automated-transformation:
        method::id result::string
```

Any transformation step is either bound to an XBGF file or relates to a generator. The latter is not necessarily a one-to-one relation, in the Java case study some scripts were designed so universally that they were re-used several times for different sources. The re-use requires higher expertise level and better accuracy in grammar manipulation, but pays off in large projects. Transformation *generators* are external tools that take a BGF grammar as an input and produce an XBGF script applicable to that grammar and containing transformations of

a certain nature, see §5.4. Generators are defined at the top-level just as transformers or comparators, so that they can be applied in different places. LCI is prepared for a generator to fail or to produce inapplicable scripts.

Whenever a generator or a script fails, that branch is terminated prematurely, implying that all consecutive transformations will fail. For all branches that reach the target, their results are compared pairwise to all others. If all branches fail or the comparator reports a mismatch, the target fails.

The graph that depicts all sources and targets as vertices and all branches as edges, is called a **convergence graph** (or a convergence tree, if it is a tree). The examples have already been provided on Figures 2 and 3.

6.1 Grammar Comparison and Measurement

If (x, y) represents sequential composition of symbols x and y, and $(x; y)$ represents a choice with x and y as alternatives, then the following formulæ are used for normalising grammars as a post-transformation or pre-comparison activity:

$$
\begin{array}{rr}
(,) \Rightarrow \varepsilon & (;) \Rightarrow fail \\
(\ldots,(x,\ldots,z),\ldots) \Rightarrow (\ldots,x,\ldots,z,\ldots) & (x,) \Rightarrow x \\
(\ldots,x,\varepsilon,z,\ldots) \Rightarrow (\ldots,x,z,\ldots) & (x;) \Rightarrow x \\
(\ldots;(x;\ldots;z);\ldots) \Rightarrow (\ldots;x;\ldots;z;\ldots) & \varepsilon^{+} \Rightarrow \varepsilon \\
(\ldots;x;fail;z;\ldots) \Rightarrow (\ldots;x;z;\ldots) & \varepsilon^{\star} \Rightarrow \varepsilon \\
(\ldots;x;\ldots;x;z;\ldots) \Rightarrow (\ldots;x;\ldots;z;\ldots) & \varepsilon? \Rightarrow \varepsilon
\end{array}
$$

The output of the comparator, boiled down to the number of mismatches, is used for measuring the progress when working on huge convergence scenarios like the JLS one. We say that we face a *nominal mismatch* when a nonterminal is defined or referenced in one of the grammars but not in the other. We face a *structural mismatch* when the definitions of a shared nonterminal differ. For every nonterminal, we count the maximum number of unmatched alternatives (of either grammar) as the number of structural mismatches [16].

The Levenshtein distance and similar clone detection metrics used for code analysis [21] can be applied to grammars. The result of such comparison can possibly be suggestive enough for automated generation of transformation scripts—this is our future work in progress at the moment. There is significant related work on schema matching [19] and model diffing [4] as well.

6.2 Validation and Error Control

Validator is an optional tool that is asked to check the XML validity of every grammar produced in the convergence process. Normally all BGF grammars produced during convergence are valid, which means if validation fails, there is something fundamentally wrong with the extractor or another part that produced it. The LCI is ready for any external tool to fail or behave inappropriately. For example, the generators discussed in the previous section can fail; can produce invalid scripts; can produce inapplicable scripts; can produce scripts that

produce invalid grammars. The error handling mechanism must be prepared for any of those possibilities and the report of the LCI should be useful for a grammar engineer.

6.3 Pretty-Printing

Strictly speaking, the presentation level itself is not a necessary part in the grammar convergence approach. However, the LCI does not exist in vacuum, and in this section we describe three most important application points for grammar representation. We call the presentation layer "pretty-printing" since it mostly comprises taking a grammar stored in its abstract form and serialising it in a specific concrete notation.

Debugging activities are unavoidable even for the best grammar engineers. When grammars are extracted, they need some kind of cursory examination that is cumbersome in pure XML. When grammars are compared, the comparison results need to be presented in the most concise and the most expressive way possible. To create a comprehensive test set one needs a way to print out any particular test case in a clear form. For tasks like these in our infrastructure we have uniform pretty-printers from BGF (grammars), XBGF (transformations) and LDF (documentation).

Publishing can take a form of an example included in a paper or in a thesis, or it can be a complete hypertext manual. Somewhat more sophisticated pretty-printers are required at this stage: for instance, a language manual in LDF can be transformed to a PDF document, to an HTML web page, to a LaTeX source, to an XSL:FO sheet, etc.

Connecting to other frameworks is the most complicated form of pretty-printing. When a functionality is required by our research that is already handled by an existing framework, it is better to pretty-print the input that is expected by that framework, use the external tool, and import back the result. For instance, the DMS software engineering toolkit [1] contains much more advanced grammar comparator which we can utilise after pretty-printing BGF as DMS. Another example can be the lack of parser generation facility in our own infrastructure: the MetaEnvironment [11] can generate it for us, if we serialise BGF as SDF. (Naturally, the lexical part could not be derived and had to be added by hand).

7 Conclusion

Essentially the LCI tool set is a model-driven framework for the domain of language recovery, transformation and convergence. LCI Configuration Format is a DSL that allows a language engineer to express the domain concepts in a concise and abstract form. Several other DSLs were designed to be used for expressing grammar knowledge, transformation steps, parse trees, language documents. It has been shown both by argument and by example that utilising these DSLs helps to take on convergence scenarios of considerable size. Our future areas of research interest include both strengthening the automation aspect by providing

more generators and introducing inferred transformation steps, on one hand, and widening the application area to full-scale language convergence by working on bidirectional and coupled transformations, on the other hand.

References

1. Baxter, I.D., Pidgeon, C.W.: Software Change through Design Maintenance. In: Proceedings of the International Conference on Software Maintenance, ICSM 1997, Washington, DC, USA, p. 250. IEEE Computer Society Press, Los Alamitos (1997)
2. Bezivin, J., Jouault, F., Valduriez, P.: On the Need for Megamodels. In: Proceedings of Workshop on Best Practices for Model-Driven Software Development at the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Vancouver, British Columbia, Canada (October 2004)
3. Cordy, J.R.: The TXL Source Transformation Language. Science of Computer Programming 61(3), 190–210 (2006)
4. Falleri, J.-R., Huchard, M., Lafourcade, M., Nebut, C.: Metamodel Matching for Automatic Model Transformation Generation. In: Busch, C., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 326–340. Springer, Heidelberg (2008)
5. Gao, S., Sperberg-McQueen, C.M., Thompson, H.S., Mendelsohn, N., Beech, D., Maloney, M.: W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. In: W3C Candidate Recommendation (April 30, 2009)
6. Gosling, J., Joy, B., Steele, G.L.: The Java Language Specification. Addison-Wesley, Reading (1996)
7. Gosling, J., Joy, B., Steele, G.L., Bracha, G.: The Java Language Specification, 2nd edn. Addison-Wesley, Reading (2000)
8. Gosling, J., Joy, B., Steele, G.L., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley, Reading (2005)
9. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The Syntax Definition Formalism SDF—Reference Manual. ACM SIGPLAN Notices 24(11), 43–75 (1989)
10. JCP JSR 31. JAXB 2.0/2.1 — Java Architecture for XML Binding (2008)
11. Klint, P.: A Meta-Environment for Generating Programming Environments. ACM Transactions on Software Engineering and Methodology (TOSEM) 2(2), 176–201 (1993)
12. Klusener, S., Zaytsev, V.: ISO/IEC JTC1/SC22 Document N3977—Language Standardization Needs Grammarware (2005), `http://www.open-std.org/jtc1/sc22/open/n3977.pdf`
13. Kort, J., Lämmel, R., Verhoef, C.: The Grammar Deployment Kit. In: van den Brand, M.G.J., Lämmel, R. (eds.) Electronic Notes in Theoretical Computer Science, vol. 65. Elsevier Science Publishers, Amsterdam (2002)
14. Lämmel, R., Verhoef, C.: Semi-automatic Grammar Recovery. Software—Practice & Experience 31(15), 1395–1438 (2001)
15. Lämmel, R., Zaytsev, V.: An Introduction to Grammar Convergence. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 246–260. Springer, Heidelberg (2009)
16. Lämmel, R., Zaytsev, V.: Recovering Grammar Relationships for the Java Language Specification. In: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 178–186. IEEE, Los Alamitos (September 2009); Full version for Software Quality Journal is in print

17. Object Management Group: MOF 2.0/XMI Mapping, 2.1.1 edn. (December 2007)
18. Parr, T.: ANTLR—ANother Tool for Language Recognition (2008)
19. Rahm, E., Bernstein, P.A.: A Survey of Approaches to Automatic Schema Matching. VLDB Journal 10(4), 334–350 (2001)
20. Sellink, A., Verhoef, C.: Generation of Software Renovation Factories from Compilers. In: Proceedings of 15th International Conference on Software Maintenance (ICSM 1999), pp. 245–255 (1999)
21. Tiarks, R., Koschke, R., Falke, R.: An Assessment of Type-3 Clones as Detected by State-of-the-Art Tools. In: 9th IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 67–76. IEEE, Los Alamitos (September 2009)
22. Zaytsev, V.: XBGF Manual: BGF Transformation Operator Suite v.1.0 (August 2009), `http://slps.sf.net/xbgf`
23. Zaytsev, V., Lämmel, R.: Language Documentation: Survey and Synthesis of a Unified Format. Submitted for publication; online since (July 7, 2010)

Author Index