

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

University of Dortmund, Germany

Madhu Sudan

Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Dragan Gašević Ralf Lämmel
Eric Van Wyk (Eds.)

Software Language Engineering

First International Conference, SLE 2008
Toulouse, France, September 29-30, 2008
Revised Selected Papers



Springer

Volume Editors

Dragan Gašević
Athabasca University
Athabasca, AB T9S 3A3, Canada
E-mail: dragang@athabascau.ca

Ralf Lämmel
Universität Koblenz-Landau
56070 Koblenz, Germany
E-mail: rlaemmel@acm.org

Eric Van Wyk
University of Minnesota
Minneapolis, MN 55455, USA
E-mail: evw@cs.umn.edu

Library of Congress Control Number: Applied for

CR Subject Classification (1998): D.2, D.3, I.6, F.3, K.6.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN	0302-9743
ISBN-10	3-642-00433-4 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-00433-9 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

springer.com

© Springer-Verlag Berlin Heidelberg 2009
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12626892 06/3180 5 4 3 2 1 0

Preface

We are pleased to present the proceedings of the First International Conference on Software Language Engineering (SLE 2008). The conference was held in Toulouse, France during September 29-30, 2008 and was co-located with the 11th IEEE/ACM International Conference on Model-Driven Engineering Languages and Systems (MODELS 2008). The SLE conference series is devoted to a wide range of topics related to artificial languages in software engineering. SLE is an international research forum that brings together researchers and practitioners from both industry and academia to expand the frontiers of software language engineering. Historically, SLE emerged from two established workshop series: LDTA, Language Descriptions, Tools, and Applications, which has been a satellite event at ETAPS for the last 9 years, and ATEM which has been co-located with MODELS and WCRE for 5 years.

SLE's foremost mission is to encourage and organize communication between communities that have traditionally looked at software languages from different, more specialized, and yet complementary perspectives. SLE emphasizes the fundamental notion of languages as opposed to any realization in specific technical spaces. In this context, the term “software language” comprises all sorts of artificial languages used in software development including general-purpose programming languages, domain-specific languages, modeling and meta-modeling languages, data models, and ontologies. Software language engineering is the application of a systematic, disciplined, quantifiable approach to the development, use, and maintenance of these languages. The SLE conference is concerned with all phases of the lifecycle of software languages; these include the design, implementation, documentation, testing, deployment, evolution, recovery, and retirement of languages. Of special interest are tools, techniques, methods, and formalisms that support these activities. In particular, tools are often based on, or automatically generated from, a formal description of the language. Hence, the treatment of language descriptions as software artifacts, akin to programs, is of particular interest—while noting the special status of language descriptions and the tailored engineering principles and methods for modularization, refactoring, refinement, composition, versioning, co-evolution, and analysis that can be applied to them.

The response to the call for papers for SLE 2008 was quite enthusiastic. We received 90 full submissions from 106 initial abstract submissions. From these 90 submissions, the Program Committee (PC) selected 18 papers: 16 full papers, one short paper, and one tool demonstration paper, resulting in an acceptance rate of 20%. To ensure the quality of the accepted papers, each submitted paper was reviewed by at least three PC members. Each paper was discussed in detail during the week-long electronic PC meeting. A summary of this discussion was

prepared by members of the PC and provided to the authors along with the reviews.

The conference was quite interactive, and the discussions provided additional feedback to the authors. Accepted papers were then revised based on the reviews, the PC discussion summary, and feedback from the conference. The final versions of all accepted papers are included in this proceedings volume. The resulting program covers diverse topics related to software language engineering. The papers cover engineering aspects in different phases of the software language development lifecycle. These include the analysis of languages in the design phase and their actual usage after deployment as well as various tools and techniques used in language implementations including different approaches to language transformation and composition. The organization of these papers in this volume reflects the sessions in the original program of the conference. SLE 2008 also had two renowned keynote speakers: Anneke Kleppe and Mark van den Brand. They each provided informative and entertaining keynote talks. Kleppe's keynote emphasized the importance of the development of general principles for the emerging discipline of software language engineering. van den Brand's keynote discussed research experiences in applying general language technology to model-driven engineering. The proceedings begin with short papers summarizing the keynotes to provide a broad introduction to the software language engineering discipline and to identify key research challenges.

SLE 2008 would not have been possible without the significant contributions of many individuals and organizations. We are grateful to the organizers of MODELS 2008 for their close collaboration and management of many of the logistics. This allowed us to offer SLE participants the opportunity to take part in two high-quality research events in the domain of software engineering. We also wish to thank our sponsors, Ateji, Capgemini, Obeo, and Software Improvement Group, for their financial support. The SLE 2008 Organizing Committee, the Local Chairs, and the SLE Steering Committee provided invaluable assistance and guidance. We are also grateful to the PC members and the additional reviewers for their dedication in reviewing the large number of submissions. We also thank the authors for their efforts in writing and then revising their papers, and we thank Springer for publishing the papers. Finally we wish to thank all the participants at SLE 2008 for the energetic and insightful discussions that made SLE 2008 such an educational and fun event.

By all indications, SLE 2008 was a very successful event. We hope that it is just the first of what will be many more successful instances of this conference and that it has fostered fruitful interactions between researchers in different software language engineering communities.

December 2008

Dragan Gašević
Ralf Lämmel
Eric Van Wyk

Organization

SLE 2008 was organized by Athabasca University, Universität Koblenz-Landau, and the University of Minnesota. It was sponsored by Ateji, Capgemini, Obeo, and Software Improvement Group.

General Chair

Ralf Lämmel	Universität Koblenz-Landau, Germany
-------------	-------------------------------------

Program Committee Co-chairs

Dragan Gašević	Athabasca University, Canada
Eric Van Wyk	University of Minnesota, USA

Organizing Committee

Jean-Marie Favre	University of Grenoble, France
Jen-Sebastien Sottet	University of Grenoble, France (Website Chair)
Andreas Winter	Johannes Gutenberg-Universität Mainz, Germany
Steffen Zschaler	Technische Universität Dresden, Germany (Publicity Chair)

Local Chairs

Benoit Combamale	University of Toulouse, France
Daniel Hagimont	University of Toulouse, France

Program Committee

Uwe Aßmann	Technische Universität Dresden, Germany
Colin Atkinson	Universität Mannheim, Germany
Jean Bezin	Université de Nantes, France
Judith Bishop	University of Pretoria, South Africa
Marco Brambilla	Politecnico di Milano, Italy
Martin Bravenboer	University of Oregon, USA
Charles Consel	LaBRI / INRIA, France
Torbjörn Ekman	Oxford University, UK
Gregor Engels	Universität Paderborn, Germany
Robert Fuhrer	IBM, USA
Martin Gogolla	University of Bremen, Germany

Jeff Gray	University of Alabama at Birmingham, USA
Klaus Havelund	NASA Jet Propulsion Laboratory, USA
Reiko Heckel	University of Leicester, UK
Nigel Horspool	University of Victoria, Canada
Joe Kiniry	University College Dublin, Ireland
Paul Klint	CWI, The Netherlands
Mitch Kokar	Northeastern University, USA
Thomas Kühne	Victoria University of Wellington, New Zealand
Julia Lawall	DIKU, Denmark
Oege de Moor	Oxford University, UK
Pierre-Etienne Moreau	INRIA & LORIA, France
Pierre-Alain Muller	University of Haute-Alsace, France
Richard Paige	University of York, UK
Jeff Pan	University of Aberdeen, UK
João Saraiva	Universidade do Minho, Portugal
Michael Schwartzbach	University of Aarhus, Denmark
Anthony Sloane	Macquarie University, Australia
Steffen Staab	Universität Koblenz-Landau, Germany
Laurence Tratt	Bournemouth University, UK
Walid Taha	Rice University, USA
Eli Tilevich	Virginia Tech, USA
Juha-Pekka Tolvanen	MetaCase, Finland
Jurgen Vinju	CWI, The Netherlands
Mike Whalen	Rockwell Collins, USA
Steffen Zschaler	Technische Universität Dresden, Germany

Additional Reviewers

Emilie Balland	László Gönczy	Kirsten Mewes
Andreas Bartho	Ulrich Hannemann	Fiona Polack
Dénes Bisztray	Karsten Hoelscher	Yuan Ren
Artur Boronoat	Karsten Hölscher	Sebastian Richly
Marko Boskovic	Nophadol Jekjantuk	Louis Rose
Paul Brauner	Jendrik Johannes	Suman Roychoudhury
Jose Creissac Campos	Sven Karol	Cherif Salama
Damien Cassou	Dimitrios Kolovos	Tim Schattkowsky
Dominique Colnet	Radu Kopetz	Mirko Seifert
Duc-Hanh Dang	Mirco Kuhlmann	Maria Semenyak
Zekai Demirezen	Carlos Matos	Yu Sun
Alexander Dotor	Julien Mercadal	Andreas Wübbecke
Nikos Drivalos	Marjan Mernik	
Alexander Foerster		

Steering Committee

Mark van den Brand	Technische Universiteit Eindhoven, The Netherlands
James Cordy	Queen's University, Canada
Jean-Marie Favre	University of Grenoble, France
Dragan Gašević	Athabasca University, Canada
Görel Hedin	Lund University, Sweden
Ralf Lämmel	Universität Koblenz-Landau, Germany
Eric Van Wyk	University of Minnesota, USA
Andreas Winter	Johannes Gutenberg-Universität Mainz, Germany

Sponsoring Institutions



Ateji, Paris, France



Capgemini, Utrecht, The Netherlands



Obeo, Rezé, France



Software Improvement Group, Amsterdam,
The Netherlands

Table of Contents

I Keynotes

The Field of Software Language Engineering	1
<i>Anneke Kleppe</i>	
Model-Driven Engineering Meets Generic Language Technology	8
<i>M.G.J. van den Brand</i>	

II Regular Papers

Session: Language and Tool Analysis and Evaluation

Evaluating the Visual Syntax of UML: An Analysis of the Cognitive Effectiveness of the UML Family of Diagrams.....	16
<i>Daniel Moody and Jos van Hillegersberg</i>	
Neon: A Library for Language Usage Analysis	35
<i>Jurriaan Hage and Peter van Keeken</i>	
Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude	54
<i>José Eduardo Rivera, Esther Guerra, Juan de Lara, and Antonio Vallecillo</i>	

Session: Concrete and Abstract Syntax

Parse Table Composition: Separate Compilation and Binary Extensibility of Grammars	74
<i>Martin Bravenboer and Eelco Visser</i>	
Practical Scope Recovery Using Bridge Parsing	95
<i>Emma Nilsson-Nyman, Torbjörn Ekman, and Görel Hedin</i>	
Generating Rewritable Abstract Syntax Trees: A Foundation for the Rapid Development of Source Code Transformation Tools	114
<i>Jeffrey L. Overbey and Ralph E. Johnson</i>	

Session: Language Engineering Techniques

Systematic Usage of Embedded Modelling Languages in Automated Model Transformation Chains	134
<i>Mathias Fritzsche, Jendrik Johannes, Uwe Aßmann, Simon Mitschke, Wasif Gilani, Ivor Spence, John Brown, and Peter Kilpatrick</i>	
Engineering a DSL for Software Traceability	151
<i>Nikolaos Drivalos, Dimitrios S. Kolovos, Richard F. Paige, and Kiran J. Fernandes</i>	
Towards an Incremental Update Approach for Concrete Textual Syntaxes for UUID-Based Model Repositories	168
<i>Thomas Goldschmidt</i>	
A Model Engineering Approach to Tool Interoperability	178
<i>Yu Sun, Zekai Demirezen, Frédéric Jouault, Robert Tairas, and Jeff Gray</i>	

Session: Language Integration and Transformation

Engineering Languages for Specifying Product-Derivation Processes in Software Product Lines	188
<i>Pablo Sánchez, Neil Loughran, Lidia Fuentes, and Alessandro Garcia</i>	
Transformation Language Integration Based on Profiles and Higher Order Transformations	208
<i>Pieter Van Gorp, Anne Keller, and Dirk Janssens</i>	
Formalization and Rule-Based Transformation of EMF Ecore-Based Models	227
<i>Bernhard Schätz</i>	

Session: Language Implementation and Analysis

A Practical Evaluation of Using TXL for Model Transformation	245
<i>Hongzhi Liang and Juergen Dingel</i>	
DEFACTO: Language-Parametric Fact Extraction from Source Code	265
<i>H.J.S. Basten and P. Klint</i>	
A Case Study in Grammar Engineering	285
<i>Tiago L. Alves and Joost Visser</i>	

Session: Language Engineering Pearls

Sudoku – A Language Description Case Study 305
 Terje Gjørøster, Ingelin F. Isfeldt, and Andreas Prinz

The Java Programmer’s Phrase Book 322
 Einar W. Høst and Bjarte M. Østfold

Author Index 343

The Field of Software Language Engineering

Keynote to the First International Conference on Software Language Engineering

Anneke Kleppe

Capgemini Netherlands
Papendorpseweg 100, Utrecht, Netherlands
a.kleppe@capgemini.com

Abstract. This paper is a written account of the keynote presented at the first International Conference on Software Language Engineering (SLE). Its key message is that although SLE is a new scientific field, we need to use existing knowledge from other fields. A number of research assignments are recognised, which are key research questions into software language design and use.

Keywords: language engineering, software language, mogram, linguistic utterance, abstract syntax, concrete syntax, semantics, keynote.

1 Introduction

This paper is a written account of the keynote presented at the first International Conference on Software Language Engineering held in Toulouse, France in September 2008. Software Language Engineering (SLE) is a new scientific field, a new patch in the great garden of knowledge, which we do not yet know how to exploit. In order to find out how to bring this bit of garden to fruit, we need to explore it first. An account on my exploration of the field can be found in [1], a book which covers most of the corners of this patch of garden. The rest of this paper gives a short overview of the ideas covered in this book.

These explorations expose a number of inhabitants. First there are workers in the garden: the worms and bees, but there are also inhabitants that are not so hard-working: the birds and rabbits. The latter merely use the fruit of the garden, the knowledge of how to design and use software languages, but they do not participate in growing that knowledge. As such this is a different tale about birds and bees. From further examinations we can conclude that there is also a difference between the hard-working inhabitants. The worm digs deep into a small part of the garden thus improving the humidity and structure of the soil, whereas the bee flies around to visit and fertilize many plants. Currently scientific research is performed by many worms, but few bees.

The audience attracted to a conference on SLE is likely to consist of bees and worms. Because they are getting ready to work in a new area of research, they might not know how to proceed, how to bring the garden of SLE to bloom. This paper gives an overview of the topics that play a part in research into SLE, thus charting the map for this new territory. A number of research questions are raised, and some directions

for answering them are given. Furthermore, this paper encourages the work of the bees, taking the knowledge that exists in other field of expertise and applying that to SLE.

In creating a new software language only three questions are important: how to specify what is a correct phrase; how to show the language's phrases to its users; and how to specify the meaning of the phrases. The first is answered by the language's abstract syntax, the second by the concrete syntax(es), the third by the semantics. This paper is structured according to these three questions, but first, in Section 2, the concept software language, which is so fundamental to SLE, is defined. Combined with the concept of software language the concept of linguistic utterance or mogram is explained. Section 3, 4 and 5 explain the abstract syntax, concrete syntax, and semantics, respectively. Some separate issues concerning the pragmatics of software languages are described in Section 6. Section 7 contains an epilogue with an entreaty to have much more bee-like researchers in the area of SLE.

2 Software Languages

In the research field of SLE there is one fundamental question to be answered: what is a language? One can safely state that every language is jargon that is spoken in a certain domain. See for example the phrases in Figure 1 that are part of a dictionary on coffee jargon (from <http://www.frontporchcoffeeandtea.com/coffeebasics.html>). Note that every phrase consists of a few words that indicate a complex concept. Thus a language is a set of words and phrases representing closely-knit, complex concepts.

A more formal, but excellent definition of the concept language is given in formal language theory (see for instance [2]). I adhere to this definition, although stated slightly different.

Definition 1 (*Language*). A language L is a set of linguistic utterances.

Obviously, this definition does not make sense until I define the concept of *linguistic utterance*. This term stems from natural language research where it represents any expression in a certain language, for instance a word, a sentence, or a conversation [3,4].

Americano. A shot of espresso with hot water added, a rich full bodied taste

Depth Charge or Shot in the Dark. Shot of espresso added to brewed coffee

Drip Coffee. Regular coffee

Espresso. Both a drink and a brewing method for coffee using pressurized hot water to extract the full flavour from a ground, specialty "espresso roast" coffee. The extraction should take at least 18, but not more than 22 seconds to produce the best flavour and crema.

Split Shot. A shot of espresso made with half regular and half decaf

With Room. Space left at top of a cup for adding cream, milk or sweetener

Fig. 1. Phrases from a dictionary on coffee jargon

The next question that needs to be answered is: what makes a language a *software* language? I define software language as follows.

Definition 2 (*Software Language*) . A *software language* is a language that is created with the purpose to describe and create software systems.

A software language is different from a natural language in that it is an artificial language, i.e. it is purposely created by some person or persons. Natural languages evolve gradually, in an analogue and free manner. Every person may create new words, and no one has the absolute authority to either reject or introduce certain terms. Artificial languages, on the other hand, change in a more digital and rigid manner. A certain group of people responsible for creating the compiler and/or the official specification decides which phrases are valid and which are not. A language user is not capable of introducing new phrases into the language. Furthermore, software languages evolve in a digital fashion: from version to version.

To make absolutely clear that the above definitions can be used not only for modelling languages, but for any language, I introduced the term *mogram* in [1], to be used instead of linguistic utterance. A mogram can be either a model or a program, a database schema or query, an XML file, or any other thing written in a software language.

2.1 Jargon about Software

When a language can be characterised as jargon, a software language can be characterised as jargon about software. However, as section 2.2 will show, software languages can also be jargon about some business domain.

Software languages that target the software domain are a special type of domain specific language (DSL), which are also called horizontal, technical, or broad DSLs. Their user group consists of IT experts. Their phrases represent complex concepts from the domain of software, like object, class, proxy, pattern. Technical software languages provide single words or small phrases for each of these complex concepts.

2.2 Jargon about Business

There are also software languages can be regarded as jargon about some business domain. These are also called vertical, business, or narrow DSLs. Their user group consists of business experts. Their phrases represent complex concepts from the given business domain, like pension plan, dossier, or tax income. Business software languages provide single words or small phrases for each of these complex, business concepts.

The reason that these language are still considered to be software languages is that a mogram written in a business DSL is translated in some form of software. Either it is directly executed, for instance in a simulation of a business process, or it is transformed into an application that provides support for the given business domain, for instance by supporting the definition of new insurance products.

3 Abstract Syntax: What Are Correct Phrases?

Because software languages are artificial languages, there needs to be a definitive guide on which phrases are considered to be valid and which are not. This definitive

guide is one of the parts of a language specification and is called the abstract syntax model (ASM). Each ASM defines the types of the words in the language and the possible (valid) relationships between those words. For instance, both class and association are types of words in the UML. An extra rule states their relationship: an association must be connected to at least two classes. It is common for an ASM to contain both definitions of types and rules that holds over these types.

There is a lot of knowledge available on how to create an ASM, for example in the following domains: domain modelling, meta modelling, data modelling, and ontologies. Therefore, the first research assignment to people willing to grow the field of SLE is to know and map the meta ontologies on each other, i.e. compare the terminology used in all these domains. For instance, do the following sets of terms have the same meaning and, if not, what are the differences?

- Universals / Types / Classes
- Particulars / Individuals / Objects / Instances / Entities
- Restrictions / Constraints
- Attributes/ Properties / Features
- Abstract Syntax Graph / Abstract Syntax Tree / Instance of a Metamodel

4 Concrete Syntax: How to Show the Words and Phrases?

Once we know the abstract syntax of the language, the next decision of the language designer is how to represent the phrases to the language user: the concrete syntax. In the coffee jargon example, the representation is taken from the English language. Likewise, when the abstract syntax is defined as a UML profile, the concrete representation of the phrases in that language follow the symbols and layout of the concrete representation of the UML. But not all languages depend on another for their concrete representation, most of them define their own concrete syntax model. Furthermore, there are languages that have multiple concrete representations, for example, a “normal” and an interchange format.

An important distinction between concrete syntaxes is whether the language is represented textually or graphically (visually). Again, a lot of existing knowledge is present to help the SLE researcher. Textual representations were studied in the areas of grammars and parser generators. Graphical language were studied by people from the visual languages community, and from the graph grammar community, and by people interested in user interface design. A number of guidelines on concrete representations from these sources are very alike:

- Use colour sparsely
- Cater for small mograms
- Keep the mogram small or above the ‘fold’, i.e. visible to the user without scrolling
- Do not repeat context, for instance, try to remove the reoccurrence of *myFav* in the following code snippet.

```
Espresso myFav = new Espresso();
myFav.size(double);
myFav.sugar(none);
```

A second research assignment for the SLE community is to capture the guidelines from various sources and indicate which guidelines are universal, and which apply to either textual or graphical languages only.

5 Semantics: How to Define Meaning?

The third aspect of a software language is its semantics: what is the meaning of the phrases in the language? I define semantics as follows:

Definition 3 (*Semantics Description*) . A description of the semantics of a language L is means to communicate a subjective understanding of the linguistic utterances of L to another person or persons.

What is important to understand is that a computer is never part of the audience of a semantics description, because a computer does not have a subjective understanding of the language's mograms. Semantics are present for human consumption only. There are several ways to describe semantics:

- Denotational, that is by constructing mathematical objects (called *denotations* or *meanings*) which represent the meaning of the mogram.
- Operational, that is by describing how a valid mogram is interpreted as sequences of computational steps. The sequence of computational steps is often given in the form of a *state transition system*, which shows how the runtime system progresses from state to state.
- Translational, that is by translating the mogram into another language that is well understood.
- Pragmatic, that is by providing a tool that executes the mogram. This tool is often called a *reference implementation*.

In my view, the best way to describe semantics to an audience of computer scientists is either translational or operational. If you can find a good target language, then the translational approach is probably the best, and certainly the quickest. However, when you lack a good target language, the operational approach is second best.

Again, existing knowledge is widely available, for instance in the areas of formal and denotational semantics, in experience with compiler technology, code generators, and software factories. From these areas I choose a single one to formulate yet another research assignment to the SLE community: how to cost-effectively implement a code generator. Many other research assignments can be found. Actually, some questions like what are the advantages of compilation/generation versus interpretation, still appear to be interesting research topics to people who are not aware of existing knowledge. But often these questions have already been answered satisfactory in earlier years. A comparison of compilation/generation and interpretation can for instance be found in [5].

6 Other Important Issues

Once the three parts of a language specification are satisfactory completed, some issues remain. At first glance these issues appear to be of a practical nature, but further examination shows that these issues, too, deserve attention from a more theoretical side.

6.1 Language Usage

In natural language studies there is a field called pragmatics which deals with how languages are being used. In the study of software languages this subject should not be skipped. Three separate aspects come to mind immediately.

Libraries and Frameworks

Language designers are often focused on creating the language specification (or worse: on a single part of the language specification). Yet a very important prerequisite for getting the language used is to provide a good library or set of libraries and frameworks. However, there is no clear evidence of what constitutes a good library or framework and where to draw the line between incorporating concepts in the abstract syntax and providing them as predefined library element.

Learn from experiences with reuse

For many years reuse has been an unreachable goal in software engineering. Getting a language and its libraries and frameworks used is very similar to reuse of arbitrary software artifacts. Much knowledge on good and not so good practices with regard to reuse is available and should be studied by the SLE researcher.

Best practices

A third aspect of the pragmatics of software languages is how to identify the best way to use the language. Specially, when the language is a newly created DSL it is not easy to provide your users with some guidance. Investigations need to be done into ways of discovering these best practices early in the lifetime of a DSL.

6.2 Polyglot Mogramming

There is no denying that currently most software system are build using many languages and many files containing source code, mappings, parameter settings, and so on. Thus, an issue that certainly deserves more attention from a theoretical point of view is that of polyglot mogramming, or creating a software system with multiple, separate mograms written in multiple languages, some domain specific and some general languages. These mograms need to contain soft references to each other, in the same way that the type of a method parameter in a Java class references another class, only in this case the references extend beyond the language. What is the effect of polyglot mogramming on tooling?

6.3 Combining Editing and Code Generations

Code generation seems to be a good choice to provide a semantics to a DSL. However, these semantics need to be recognizable to the language user in an early

stage - preferably during editing of the program. How can we build a tool chain from concrete to abstract syntax graph in which these semantics (and especially errors against the semantical rules) can be presented in an intuitive manner to the language user? When the (errors against the) semantics are dynamic of nature, i.e. only tangible when the system is running in for instance a debugger, how do we represent them? What are the best practices in cost-effectively creating a debugger?

7 Epilogue

In this paper I have explored the scope of the field of new scientific research called software language engineering. Whenever we call something new, we should realize that hardly ever there is something that is really complete new. More likely, it will be the case that the new consists of old things reassembled, as is gracefully described by novelist Connie Palmen in [6]:

"... (to) make something that was not there before, ... is deceptive, because the separate elements already existed and floated through history, but they were never before assembled in this manner. Joining, assembling, the new will always consist of that." (Italics by AK)

In this paper I have indicated what I feel are the most important elements of SLE, and the most pertinent research questions at this moment in time. To answer these questions we can draw from many other scientific fields of knowledge, but for this we need more bee-like scientist that are capable of crossing the borders between these fields, understanding the essentials and translating them into the jargon used in another field.

My hopes are that the garden called software language engineering will blossom and that the Conference of Software Language Engineering will see many happy returns!

References

- [1] Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley Longman Publishing Co., Inc., Boston (2009)
- [2] Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
- [3] Tserdanelis, G., Wong, W.Y.P. (eds.): The Language Files: Materials for an Introduction to Language and Linguistics, 9th edn. Ohio State University Press (April 2004)
- [4] Fromkin, V., Rodman, R., Hyams, N. (eds.): An Introduction to Language, 7th edn. Heinle (August 2002)
- [5] Klint, P.: Interpretation techniques. *Softw., Pract. Exper.* 11(9), 963–973 (1981)
- [6] Palmen, C.: *Lucifer*. Prometheus, Amsterdam (2007)

Model-Driven Engineering Meets Generic Language Technology

M.G.J. van den Brand

Technical University Eindhoven, Department of Mathematics and Computer Science,
Den Dolech 2, NL-5612 AZ Eindhoven, The Netherlands
`m.g.j.v.d.brand@tue.nl`

Abstract. One of the key points of model-driven engineering is raising the level of abstraction in software development. This phenomenon is not new. In the sixties of the previous century, the first high-level programming languages were developed and they also increased the abstraction level of software development. The development of high-level programming languages initiated research on compilers and programming environments. This research eventually matured into generic language technology: the description of (programming) languages and tooling to generate compilers and programming environments. The model-driven engineering community is developing tools to analyze models and to transform models into code. The application of generic language technology, or at least the lessons learnt by this community, can be beneficial for the model-driven engineering community. By means of a number of case studies it will be shown how generic language technology research can be useful for the development of model-driven engineering technology.

1 Introduction

Software engineering is currently facing two major challenges. The first challenge is to deal with the increasing required level of dependability of software. Software is more and more applied in daily life products, such as mobile phones, cars, medical equipment, and consumer electronics. A number of these application areas ask for an extremely high level of dependability. The second challenge is dealing with the increasing amount of software that has to be produced and maintained. The number of lines of code is continuously increasing. The speed of producing reliable software has to increase. The application of model-driven engineering could be a valid alternative to increase the overall productivity of software engineers.

One of the key points of model-driven engineering is the raise of abstraction. Software is modeled on a higher level of abstraction and then advanced model transformation techniques and software generation techniques are used to translate the models into executable code. Furthermore, the use of models offers the possibility to use these models for sophisticated analysis techniques such as model checking, performance analysis, correctness, etc. This allows a possible route to tackle part of the dependability problems of the first challenge.

The key concepts in model-driven engineering are the development of formalisms to develop the models, the development of model transformation engines, so-called model-to-model transformations, and the development of software generators, so-called model-to-text transformations. In this paper we show how techniques developed in the field of compiler construction can be applied to the field of model-driven engineering. We will demonstrate this by means of a number of case studies where we applied compiler construction technology to model-driven engineering problems.

2 Generic Language Technology

Computer science and compiler construction have a very close marriage. The development of programming languages, and thus the development of software, has been facilitated by developments and research in the field of compiler construction. Compilers facilitate the development of software at a higher level of abstraction. In the sixties and seventies of the previous century a lot of compiler construction related research was carried out. The generation of compilers based on language descriptions is one of the research fields of compiler construction. The development of a compiler for a specific programming language is a considerable amount of work. Compilers were and still are mainly developed manually. However, large parts of a compiler are almost identical for every programming language. The definition of specification languages for the description of programming languages and the development of tools to generate from these language descriptions parts of compilers is the focus of generic language technology.

The most important milestones in the field of generic language technology are scanner and parser generators and attribute grammars [1]. Scanner and parser generators are tools for generating scanners and parsers from lexical and context-free grammar specifications of (programming) languages. There exists a whole range of scanner and parser generator tools, e.g. Lex+Yacc [26,22], ANTLR [27], JavaCC [15], and SDF [19]. The development and maintenance of (context-free) grammars is an underestimated effort [11,24]. Grammars are not developed with reuse in mind. Exchanging grammars between different tooling is tedious and error prone. Attribute grammars are a collection of formalisms to describe the semantics of programming languages. There exist several implementations of attribute grammars, e.g. JastAdd [17], ELI [18], LISA [20]. However, exchanges of descriptions of one attribute grammar system to another is not straightforward. This diversity has not contributed to the popularity of attribute grammars and prevented it from becoming a mainstream compiler construction technology.

In the eighties, the focus of generic language technology shifted from compilers to programming environments. Starting from language descriptions, (parts of) interactive programming environments were generated. The development of software within these interactive environments was supported by syntax directed editors, type checkers, interpreters, pretty printers, and debuggers. These tools were made available to the programmer via a uniform graphical user interface.

The goal of these environments was to inform users in a very early stage on syntactic and semantic errors in the software under development. Various research groups have worked on the development of programming environment generators. The most well-known programming environment generators are the Synthesizer Generator [28] and the ASF+SDF Meta-Environment [8]. The focus of current research is on the integration of generic language technology into frameworks such as Eclipse [9] via IMP [14]. A previous attempt in this direction was SAFARI [13].

3 Model-Driven (Software) Engineering

Models have a central role in modern software development. The way software is being developed in the last sixty years has gone through quite a number of stages. The challenge was to increase the abstraction level of software development in order to increase the development efficiency and quality of the resulting software. In generic language technology this increase of abstract was done via the introduction of specification formalisms to describe programming languages. In ordinary software development there was a shift from developing software in machine code to developing software in high level programming languages. Nowadays, a lot of software is developed using domain specific languages or using the Unified Modeling Language (UML) of the OMG¹. UML was created via the unification of a number of modelling languages, initially the Object Modeling Technique [29], the Booch method [7], and use cases [21]. Later, more modeling languages have been added. The current version, UML2.0, consists of thirteen different types of diagrams. UML has become the de-facto standard of the software industry.

Meta models allow a definition of models. Meta models play the same role for models as (E)BNF for programming languages. Meta models themselves are described using meta meta models. Diagrams provide a view on (parts of) the models. See [6] for a description of these hierarchies and [25] for a description of the similarities between (meta) meta modeling and formalisms to describe textual programming languages. The fact that different meta models can be described using the same meta meta model does not mean that model transformations are straightforward. The meta model describes the structure of the model, but not the semantics. This is similar to programming languages, given the (E)BNF definitions of Cobol and Java it does not mean that a source to source translation from Cobol to Java is feasible due to semantic inconsistency.

4 Case Studies

We will discuss a number of case studies where we show how generic language technology can be applied in the domain of model-driven (software) engineering.

¹ <http://www.omg.org>

4.1 Transformation of Behavioural Models Revealed a Semantic Gap

An example of semantic inconsistency between models was discovered during the development of model transformations from models, describing the behaviour of processes in warehouses, to executable code. Formalisms, like Chi [4], based on process algebra are very well suited to model the processes in this type of distributed systems. We developed a meta model, as an SDF definition [19], for the process algebra ACP [5] and realized a model transformation from ACP to UML state machine diagrams in ASF+SDF [16]. These state machine diagrams are used to describe (concurrent) behaviour. Rhapsody² can translate this type of diagrams into executable code. From a syntactic point of view, the transformation from ACP to UML state machines is possible. However, given the requirement for Rhapsody on preserving the structure of the original specification, quite some effort is needed to get a semantic fit.

ACP consists of a number of operators, amongst others `."` (sequential composition), `."` (alternative composition), `||"` (parallel composition), and `|"` (communication). The operands can be, among others an action `"a"`, and deadlock `"δ"`. The parallel operator in ACP describes more than just the behaviour of two parallel processes. The arguments of the parallel operator can, if the specification allows this, interact and *melt* into one action. In general, ACP specifications are normalized and the result is an ACP specification where all parallel operators have disappeared and only the result of the interaction of processes is kept. These normalized specifications do not describe the original structure of the modeled system, which is crucial whenever we want to have a distribution of the software over several processors (that communicate with each other). This semantic gap can be bridged by generating next to the UML state machine diagram an execution environment which takes care of the steps performed by the state machine. The details of our solution and transformation are described in [2], in this paper also problems related to the alternative composition operator are discussed.

4.2 Surface Language as Alternative for Graphical Language

This case study is also related to embedded software development. The goal was to model the behaviour of systems in UML and use these UML specifications for performance analysis. In order to be able to do the performance analysis, the behavioural UML models are translated into POOSL [30] models. However, it turned out that a straightforward translation was not feasible.

Therefore, it was decided to investigate how POOSL could be merged with UML, which made the transformation from the extended UML models to POOSL code feasible. Initially, the focus was on defining subset of UML activities (with an imperative flavour) to enable a simple direct transformation to POOSL or some other object oriented or imperative language. This turned out to be possible, but developing behavioural specifications graphically turned out to be

² <http://www.telelogic.com>

tedious. We continued our research in the direction of developing a textual alternative for the UML activity diagrams, a so-called surface language. The development of this surface language was not the actual challenge but the combination of having, for instance, UML state machines annotated with pieces of surface language code was. We tackled this problem by combining the grammar for XMI with the grammar of the surface language, both written in SDF. The Eclipse tooling allows the export of XMI with embedded surface language. Via ASF+SDF we were able to parse this XMI, rewrite the embedded surface language code fragments into full UML activities and produce the corresponding XMI again. This resulting XMI can be processed using the Eclipse tooling.

4.3 Syntax Safe Templates

The ultimate goal of model-driven software engineering is the generation of executable code. There are several approaches for generating code from models. In fact, every tool has its own approach, for instance in openArchitectureWare³ this is done via Xpand. We investigated how we could improve the reliability of the software generators based on templates.

There are various types of software generators, see [31] for an overview. We will only consider the so-called template-based generators. This type of software generator is based on strings, it consists of an evaluator and templates describing the code that has to be generated. There is a strict separation between the evaluator and the templates. A template is fragment of object code with embedded meta language constructs. These embedded meta language constructs are interpreted by the evaluator and replaced by information obtained from input data. By combining the grammar of the object code with the grammar of the embedded meta language we are able to ensure the syntactical correctness of the generated code. We use the power of the SDF toolset, modularization of grammars, to be able to parse the templates and check for their syntactical correctness. The evaluator checks whether the input obtained from the input data is syntactic correct replacement for the meta language construct.

Our Repleo approach allows us to check templates written in Java, C, and HTML. Furthermore, it allows us to check us templates which use both Java and for instance embedded SQL. Our approach guarantees the syntax safety of a broad range of templates, based on the concept of "the more grammar, the more checking". Details on the template evaluator Repleo, can be found in [3].

Repleo has been used in an industrial project at Ericsson to translate state machines into Java code [12]. Furthermore, to validate the expressive power of Repleo the Java part of the tool ApiGen [23,10] has been reimplemented. Both case studies revealed the usefulness of our approach because syntax errors were discovered at an early stage.

³ <http://www.openarchitectureware.org/>

5 Lessons Learnt

The combination of generic language technology with model-driven software engineering is a very fruitful combination. In fact this was to be expected. Model-driven software engineering is about raising the level of abstraction and this is also the quest in compiler construction. The starting point may be different, i.e., graphical languages instead of textual languages, but it is all about translating and transforming.

We have shown a number of case studies where generic language technology has been successfully applied to the model-driven software engineering problems. Furthermore, we have shown that the same semantic problems may arise in the field of model-driven software engineering as in the field of source to source translations.

There are a number of general lessons that can be learnt applying the generic language technology:

- The development of domain specific modeling formalisms involves more than just the development of meta models or profiles. It also involves the development of tooling, methodology and even validation of the modeling formalism via empirical research.
- There is a huge amount of compiler construction related research from which the model-driven software engineering can learn. Reinventing the wheel is not a good idea and working together can accelerate the research in model-driven software engineering.
- There is one important lesson for both the compiler construction and the model-driven software engineering community: consider standardization but be critical of it. There are many tools developed by the compiler construction research community that do not allow the exchange of formalisms. The situation is better in the model-driven software engineering community. On the other hand, it turns out that (commercial) tool developers do not adhere to standardization in order to create a (vendor) lock-in.

Acknowledgements

I thank Jeroen Arnoldus, Marcel van Amstel, Zvezdan Protic and Luc Engelen for doing the actual work on the case studies and for proof reading this paper.

References

1. Alblas, H.: Introduction to attribute grammars. In: Alblas, H., Melichar, B. (eds.) SAGA School 1991. LNCS, vol. 545, pp. 1–15. Springer, Heidelberg (1991)
2. van Amstel, M.F., van den Brand, M.G.J., Protić, Z., Verhoeff, T.: Transforming process algebra models into UML state machines: Bridging a semantic gap? In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 61–75. Springer, Heidelberg (2008)

3. Arnoldus, J., Bijpost, J., van den Brand, M.G.J.: Repleo: a syntax-safe template engine. In: GPCE 2007: Proceedings of the 6th international conference on Generative programming and component engineering, pp. 25–32. ACM, New York (2007)
4. van Beek, D.A., Man, K.L., Reniers, M.A., Rooda, J.E., Schifflers, R.R.H.: Syntax and semantics of timed Chi. CS-Report 05–09, Department of Computer Science, Eindhoven University of Technology (2005)
5. Bergstra, J.A., Klop, J.W.: Algebra of communicating processes. In: de Bakker, J.W., Hazewinkel, M., Lenstra, J.K. (eds.) Proceedings of the CWI Symposium. CWI Monographs, vol. 1, pp. 89–138. Centre for Mathematics and Computer Science, North-Holland (1986)
6. Bézivin, J.: Model driven engineering: An emerging technical space. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 36–64. Springer, Heidelberg (2006)
7. Booch, G.: Object-Oriented Design with Applications, 2nd edn. Benjamin/Cummings (1993)
8. den van 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)
9. van den Brand, M.G.J., de Jong, H.A., Klint, P., Kooiker, A.T.: A Language Development Environment for Eclipse. In: Burke, M.G. (ed.) Eclipse Technology eXchange (eTX), pp. 61–66 (2003)
10. van den Brand, M.G.J., Moreau, P.E., Vinju, J.J.: A generator of efficient strongly typed abstract syntax trees in java. IEE Proceedings-Software 152(2), 70–78 (2005)
11. van den Brand, M.G.J., Sellink, A., Verhoef, C.: Current parsing techniques in software renovation considered harmful. In: IWPC 1998: Proceedings of the 6th International Workshop on Program Comprehension, pp. 108–117. IEEE Computer Society, Los Alamitos (1998)
12. Bruggeman, W.: Transformation of a SIP Service Model. Master’s thesis, Eindhoven University of Technology (2008)
13. Charles, P., Dolby, J., Fuhrer, R.M., Sutton Jr., S.M., Vaziri, M.: Safari: a meta-tooling framework for generating language-specific ide’s. In: OOPSLA 2006: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pp. 722–723. ACM, New York (2006)
14. Charles, P., Fuhrer, R.M., Sutton Jr., S.M.: Imp: a meta-tooling platform for creating language-specific ides in eclipse. In: ASE 2007: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 485–488. ACM, New York (2007)
15. Copeland, T.: Generating Parsers with JavaCC. Centennial Books (2007)
16. van Deursen, A., Heering, J., Klint, P. (eds.): Language Prototyping: An Algebraic Specification Approach. AMAST Series in Computing, vol. 5. World Scientific, Singapore (1996)
17. Ekman, T., Hedin, G.: The JastAdd System - modular extensible compiler construction. Science of Computer Programming 69(1-3), 14–26 (2007)
18. Gray, R.W., Heuring, V.P., Levi, S.P., Sloane, A.M., Waite, W.M.: Eli: A complete, flexible compiler construction system. Communications of the ACM 35(2), 121–130 (1992)
19. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF: Reference manual. SIGPLAN Notices 24(11), 43–75 (1989)

20. Henriques, P.R., Pereira, M.J.V., Mernik, M., Lenic, M., Gray, J., Wu, H.: Automatic generation of language-based tools using the LISA system. *IEE Proceedings - Software* 2(152) (2005)
21. Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G.: *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Publishing Company, Reading (1992)
22. Johnson, S.C.: YACC—yet another compiler-compiler. Technical Report CS-32, AT&T Bell Laboratories, Murray Hill, N.J. (1975)
23. de Jong, H.A., Olivier, P.A.: Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming* 50(4), 35–61 (2004)
24. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.* 14(3), 331–380 (2005)
25. Kunert, A.: Semi-automatic generation of metamodels and models from grammars and programs. *Electron. Notes Theor. Comput. Sci.* 211, 111–119 (2008)
26. Lesk, M.E., Schmidt, E.: LEX — A lexical analyzer generator. Technical Report CS-39, AT&T Bell Laboratories, Murray Hill, N.J. (1975)
27. Parr, T.J., Quong, R.W.: Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience* 25, 789–810 (1995)
28. Reps, T., Teitelbaum, T.: *The Synthesizer Generator: A System for Constructing Language-Based Editors*, 3rd edn. Springer, Heidelberg (1989)
29. Rumbaugh, J.E., Blaha, M.R., Premerlani, W.J., Eddy, F., Lorensen, W.E.: *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs (1991)
30. Theelen, B.D., Florescu, O., Geilen, M.C.W., Huang, J., van der Putten, P.H.A., Voeten, J.P.M.: Software/hardware engineering with the parallel object-oriented specification language. In: *MEMOCODE 2007: Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, Washington, DC, USA, pp. 139–148. IEEE Computer Society, Los Alamitos (2007)
31. Völter, M.: A Catalog of Patterns for Program Generation. In: *Eighth European Conference on Pattern Languages of Programs (EuroPLop)* (2003)

Evaluating the Visual Syntax of UML: An Analysis of the Cognitive Effectiveness of the UML Family of Diagrams

Daniel Moody and Jos van Hillegersberg

Department of Information Systems & Change Management
University of Twente, Enschede, Netherlands
d.l.moody@utwente.nl

Abstract. UML is a visual language. However surprisingly, there has been very little attention in either research or practice to the visual notations used in UML. Both academic analyses and official revisions to the standard have focused almost exclusively on semantic issues, with little debate about the visual syntax. We believe this is a major oversight and that as a result, UML's visual development is lagging behind its semantic development. The lack of attention to visual aspects is surprising given that the form of visual representations is known to have an equal if not greater effect on understanding and problem solving performance than their content. The UML visual notations were developed in a bottom-up manner, by reusing and synthesising existing notations, with choice of graphical conventions based on expert consensus. We argue that this is an inappropriate basis for making visual representation decisions and they should be based on theory and empirical evidence about cognitive effectiveness. This paper evaluates the visual syntax of UML using a set of evidence-based principles for designing cognitively effective visual notations. The analysis reveals some serious design flaws in the UML visual notations together with practical recommendations for fixing them.

1 Introduction

The Unified Modelling Language (UML) is widely accepted as an industry standard language for modelling software systems. The history of software engineering is characterised by competing concepts, notations and methodologies. UML has provided the software industry with a common language, something which it has never had before. Its development represents a pivotal event in the history of software engineering, which has helped to unify the field and provide a basis for further standardisation.

1.1 UML: A Visual Language

UML is a “visual language for visualising, specifying, constructing and documenting software intensive systems” [25]. The UML *visual vocabulary* (symbol set) is loosely partitioned into 13 diagram types, which define overlapping views on the underlying metamodel (Figure 1). So far, there has been remarkably little debate in research or

practice about the visual notations used in UML (also called *visual syntax* or *concrete syntax*). There have been many academic evaluations of UML, but most have focused on semantic aspects [e.g. 8, 26, 35]. There have also been several revisions to UML, but these have also concentrated on semantic issues, with little discussion about, or modification to, graphical conventions. The lack of attention to visual aspects is surprising given UML's highly visual nature. It is even more surprising in the light of research in diagrammatic reasoning that shows that the *form* of representations has a comparatively greater effect on their effectiveness than their *content* [14, 34]. Apparently minor changes in visual appearance can have dramatic impacts on understanding and problem solving performance [28].

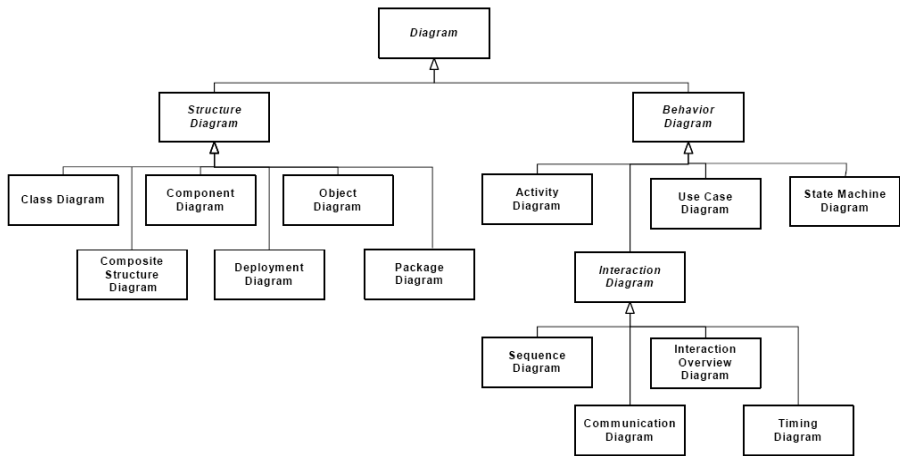


Fig. 1. The 13 diagram types partition the UML visual vocabulary into a set of overlapping sublanguages [25]

1.2 What Makes a Good Visual Language?

Diagrams are uniquely human-oriented representations: they are created *by* humans *for* humans [12]. They have little or no value for communicating with computers, whose visual processing capabilities are primitive at best. To be most effective, visual languages therefore need to be optimised for processing by the human mind. *Cognitive effectiveness* is defined as the speed, ease and accuracy with which information can be extracted from a representation [14]. This provides an operational definition for the “goodness” of a visual language and determines their utility for communication and for design and problem solving.

The cognitive effectiveness of diagrams is one of the most widely accepted assumptions in the software engineering field. However cognitive effectiveness is not an intrinsic property of diagrams but something that must be *designed* into them [14, 28]. Diagrams are not effective simply because they are graphical, and poorly-designed diagrams can be far *less* effective than text [5].

Software engineering is a collaborative process which typically involves technical experts (software developers) and business stakeholders (end users, customers). It is therefore desirable that system representations (especially at the requirements level)

can be understood by both parties: effective user-developer communication is critical for successful software development. However a common criticism of UML is its poor communicability to end users [35]. In practice, UML diagrams are often translated into textual form for verification with users, which is a clear sign of their ineffectiveness for this purpose [36].

1.3 How UML Was Developed

The graphical notations used in UML were developed in a bottom-up manner, by reusing and synthesising existing notations:

“The UML notation is a melding of graphical syntax from various sources. The UML developers did not invent most of these ideas; rather, they selected and integrated the best ideas from object modelling and computer science practices.” [25]

How the “best” ideas were identified is not explained, but seems to have been done (and still is) based on expert consensus. This paper argues that this is *not* a valid way to make graphical representation decisions, especially when the experts involved are not experts in graphic design (which is necessary to understand the implications of choices made). Being an expert in software engineering does *not* qualify someone to design visual representations. In the absence of such expertise, notation designers are forced to rely on common sense and opinion, which is unreliable: the effect of graphic design choices are often counterintuitive and our instincts can lead us horribly astray [40].

Design rationale is the process of explicitly documenting design decisions made and the reasons they were made. This helps provide traceability in the design process and to justify the final design. Design rationale explanations should include the reasons behind design decisions, alternatives considered, and trade offs evaluated [15]. Such explanations are almost totally absent from the design of the UML visual notations. Graphical conventions are typically defined by assertion (e.g. “a class is represented by a rectangle”) with no attempt to justify them.

1.4 Objectives of This Paper

We argue that the design of visual notations should be *evidence based*: choice of graphical conventions should be based on the best available research evidence about cognitive effectiveness rather than on common sense or social consensus. There is an enormous amount of research that can be used to design better visual notations (though mostly outside the software engineering field). This paper evaluates the visual syntax of UML 2.0 using a set of evidence-based principles for designing cognitively effective visual notations. Our aim is to be as constructive as possible in doing this: to improve the language rather than just point out its deficiencies. As a result, when we identify problems, we also try to offer practical (though evidence-based) suggestions for fixing them.

2 Related Research

Ontological analysis has become widely accepted as a way of evaluating the semantics of software engineering notations [7, 32]. This involves a two-way mapping between

the constructs of a notation and an ontology: the *interpretation mapping* describes the mapping from the notation to the ontology while the *representation mapping* describes the inverse mapping [7]. Ideally, there should be a one-to-one correspondence between the categories in the ontological theory and notation constructs (Figure 2). If there is not such a correspondence, one or more of the following problems will occur:

- *Construct deficit* exists when there is no semantic construct corresponding to a particular ontological category.
- *Construct overload* exists when the same semantic construct is used to represent multiple ontological categories
- *Construct redundancy* exists when multiple semantic constructs are used to represent a single ontological category
- *Construct excess* exists when a semantic construct does not correspond to any ontological category.

If construct deficit exists, the language is said to be *ontologically incomplete*. If any of the other three anomalies exist, the language is said to be *ontologically unclear*. Ontological analysis has previously been applied to evaluate the semantics of UML [26]. This paper complements this previous research by extending the evaluation of UML to the level of visual syntax (*form* rather than *content*).

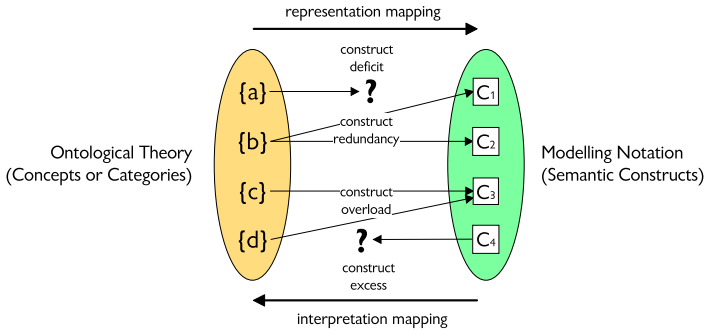


Fig. 2. Ontological Analysis: there should be a 1:1 correspondence between ontological concepts and modelling constructs

3 Principles for Visual Notation Design

A previous paper defined a set of principles for designing cognitively effective visual notations [19]. These principles were synthesised from theory and empirical evidence from a wide range of disciplines and have been previously used to evaluate and improve two other visual languages [20, 21]: *ArchiMate* [13], which has been recently adopted as an international standard for modelling enterprise architectures and *ORES*, a proprietary cognitive mapping method for organisational development and strategic planning. We use these principles in this paper as a basis for evaluating the cognitive effectiveness of the UML visual notations. This section reviews the principles.

3.1 Principle of Semiotic Clarity

This principle represents an extension of ontological analysis to the level of visual syntax using a theory from semiotics. According to Goodman's *theory of symbols* [9], for a notation to satisfy the requirements of a *notational system*, there should be a one-to-one correspondence between symbols and their referent concepts [9]. The requirements of a notational system constrain the allowable expressions and interpretations in a language in order to simplify use and minimise ambiguity: clearly, these are desirable properties for software engineering languages. When there is *not* a one-to-one correspondence between semantic constructs and graphical symbols, the following anomalies can occur (in analogy to the terminology used in ontological analysis) (Figure 3):

- *Symbol redundancy* exists when multiple symbols are used to represent the same semantic construct. Such symbols are called *synographs* (the graphical equivalent of synonyms). Symbol redundancy places a burden of choice on the language user to decide which symbol to use and an additional load on the reader to remember multiple representations of the same construct.
- *Symbol overload* exists when the same graphical symbol is used to represent different semantic constructs. Such symbols are called *homographs* (the graphical equivalent of homonyms). Symbol overload is the worst kind of anomaly as it leads to ambiguity and the potential for misinterpretation [9]. It also violates one of the basic properties of the symbol system of graphics (*monosemy* [11]), as the meaning of these symbols must emerge from the context or through the use of textual annotations.
- *Symbol excess* exists when graphical symbols are used that don't represent any semantic construct. Symbol excess unnecessarily increases *graphic complexity*, which has been found to reduce understanding of notations [23].
- *Symbol deficit* exists when semantic constructs are not represented by any graphical symbol. This is not necessarily a problem and can actually be an advantage: symbol deficit may be used as a deliberate strategy for reducing graphic complexity of notations.

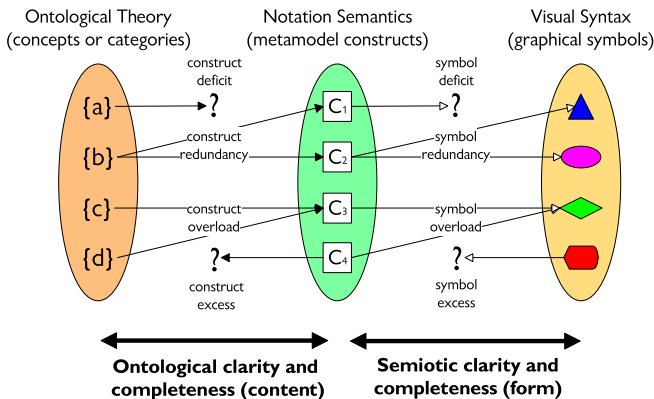


Fig. 3. Semiotic clarity represents an extension of ontological analysis to the syntactic level

If symbol deficit exists, the visual notation is said to be *semiotically incomplete*. If any of the other three anomalies exist, the notation is *semiotically unclear*.

3.2 Principle of Perceptual Discriminability

Perceptual discriminability is the ease and accuracy with which different graphical symbols can be differentiated from each other. Accurate discrimination between symbols is a necessary prerequisite for accurate interpretation of diagrams [43].

Visual Distance

Perceptual discriminability is primarily determined by the *visual distance* between symbols, which is defined by (a) the number of visual variables on which they differ and (b) the size of these differences. In general, the greater the visual distance between symbols used to represent different constructs, the faster and more accurately they will be recognised [42]. If differences are too subtle, errors in interpretation and ambiguity can result. In particular, requirements for perceptual discriminability are much higher for novices (end users) than for experts [3, 4].

Perceptual Popout

According to *feature integration theory*, visual elements that have unique values on at least one visual variable can be detected pre-attentively and in parallel across the visual field [29, 38]. Such elements “pop out” of the visual field without conscious effort. On the other hand, visual elements that are differentiated by a unique combination of values (*conjunctions*) require serial inspection to be recognised, which is much slower, error-prone and effortful [39]. The clear implication of this for visual notation design is that each graphical symbol should have a unique value on at least one visual variable.

3.3 Principle of Perceptual Immediacy

Perceptual immediacy refers to the use of graphical representations that have natural associations with the concepts or relationships they represent. While the *Principle of Perceptual Discriminability* requires that symbols used to represent different constructs should be clearly different from each other, this principle requires that symbols should (where possible) provide cues to their meaning. The most obvious form of association is perceptual resemblance, but other types of associations are possible: logical similarities, functional similarities and cultural associations.

Iconic representations

Icons are symbols which perceptually resemble the concepts they represent [27]. These make diagrams more visually appealing, speed up recognition and recall, and improve intelligibility to naïve users [3, 4]. Icons are pervasively used in user interface design [22] and cartography [44] but surprisingly rarely in software engineering.

Spatial relationships

Perceptual immediacy also applies to representation of relationships. Certain spatial configurations of visual elements predispose people towards a particular interpretation of the relationship between them even if the nature of the elements is unknown [11, 43]. For example, left-to-right arrangement of objects suggests causality or sequence while placing objects inside other objects suggests class membership (subset).

3.4 Principle of Visual Expressiveness

Visual expressiveness refers to the number of different visual variables used in a visual notation. There are 8 elementary *visual variables* which can be used to graphically encode information (Figure 4) [1]. These are categorised into *planar variables* (the two spatial dimensions) and *retinal variables* (features of the retinal image).

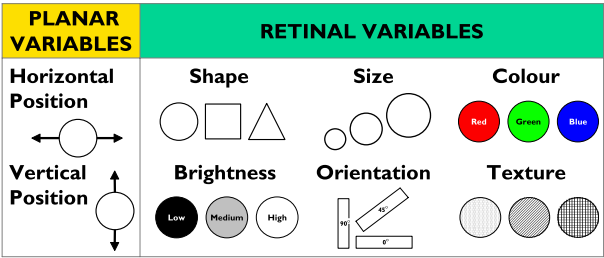


Fig. 4. The Dimensions of the Design Space: the visual variables define a set of elementary graphical techniques that can be used to construct visual notations

Using a range of variables results in a perceptually enriched representation which uses multiple, parallel channels of communication. This maximises computational off-loading and supports full utilisation of the graphic design space. Different visual variables have properties which make them suitable for encoding some types of information but not others. Knowledge of these properties is necessary to make effective choices.

3.5 Principle of Graphic Parsimony

Graphic complexity is defined as the number of distinct graphical conventions used in a notation: the size of its *visual vocabulary* [23]. Empirical studies show that increasing graphic complexity significantly reduces understanding of software engineering diagrams by naïve users [23]. It is also a major barrier to learning and use of a notation. The human ability to discriminate between perceptually distinct alternatives on a single perceptual dimension (*span of absolute judgement*) is around six categories: this defines a practical limit for graphic complexity [18].

4 Evaluation of UML

In this section, we evaluate the 13 types of diagrams defined in UML 2.0 using the principles defined in Section 3.

4.1 Principle of Semiotic Clarity

The UML visual vocabulary contains many violations to semiotic clarity: in particular, it has alarmingly high levels of symbol redundancy and symbol overload. For example, of the 31 symbols commonly used on Class diagrams (shown in Figure 6

and Table 1), there are 5 synographs (16%) and 20 homographs (65%). This results in high levels of graphic ambiguity (due to homographs) and graphic complexity (due to synographs).

- *Symbol redundancy*: this is widespread in UML: examples of synographs can be found in all diagram types. For example, in Use Case Diagrams, actors can be represented by stick figures or rectangles; in Class Diagrams, interfaces can be represented by rectangles or circles. Relationships can also be represented in different ways: for example, in Package Diagrams, package relationships can be shown using spatial enclosure or connecting lines. The purpose for providing alternative visual representations is not explained but presumably to provide flexibility to notation users. However flexibility in perceptual forms is undesirable in *any* language (e.g. alternative spellings for words in natural languages) and undermines standardisation and communication.

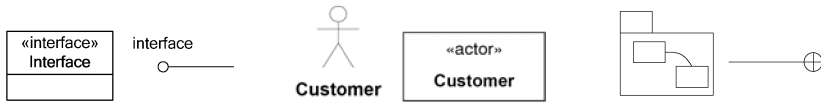


Fig. 5. Symbol redundancy: there are multiple graphical representations (*synographs*) for interfaces in Class Diagrams, actors in Use Case Diagrams and package relationships in Package Diagrams

- *Symbol overload*: this is endemic in UML, with the majority of graphical conventions used to mean different things. For example, in Class Diagrams, the same graphical symbol can be used to represent objects, classes and attributes. Different types of relationships can also be represented using the same graphical convention e.g. package merges, package imports and dependencies are all represented using dashed arrows. A major contributor to symbol overload in UML is the widespread practice of using text to discriminate between symbols (something which very few other software engineering notations do) ¹.
- *Symbol excess*: the most obvious example of symbol excess and one which affects all UML diagram types is the *note* or comment. Notes contain explanatory text to clarify the meaning of a diagram and perform a similar role to comments in programs. Including notes on diagrams is a useful practice, but enclosing them in graphical symbols is unnecessary as they convey no additional semantics. Such symbols add visual noise to the diagram (an example of what graphic designers call “boxitis” [41]) and confound its interpretation by making it likely they will be interpreted as constructs.

Recommendations for Improvement

To simplify interpretation and use of the language, all occurrences of symbol redundancy, symbol overload and symbol excess should be removed:

¹ Symbols are considered to be homographs if they have zero visual distance (i.e. they have identical values for all visual variables) but represent different semantic constructs.

- Symbol redundancy: this can be easily resolved by choosing one of the symbols as the sole representation for a construct. Two later principles (*Perceptual Discriminability* and *Perceptual Immediacy*) provide the basis for choosing between synographs (identifying the most cognitively effective alternative).
- Symbol overload: this can be resolved by differentiating between symbols used to represent different constructs. The *Principle of Perceptual Discriminability* defines ways of differentiating between symbols.
- Symbol excess: this can be resolved simply by removing the unnecessary symbols. For example, notes can be simply shown as blocks of text so they will not be interpreted as constructs. They should also be shown using smaller font so they are clearly lower in the visual hierarchy (perceptual precedence).

4.2 Principle of Perceptual Discriminability

UML diagrams consist of two types of elements: *nodes* (two-dimensional graphical elements) and *links* (one-dimensional graphical elements). Discriminability of both types of elements are important. Figure 6 shows the node types that commonly appear on Class Diagrams, which are the most important of all UML diagrams. There are several problems with the discriminability of these symbols:

- Visual proximity: the node types (with one exception) differ on only a single visual variable (*shape*) and the values chosen are very close together: all shapes are either rectangles or rectangle variants. Given that experimental studies show that rectangles and diamonds are often confused by naïve users in ER diagrams [23], it is likely that these shapes will appear virtually identical to non-experts.
- Textual encoding: some of the symbols have zero visual distance (*homographs*) and are differentiated by labels or typographical characteristics. For example, objects are distinguished from classes by use of underlining, abstract classes by *italics* and data types by a label «data type». Text is a inefficient way to differentiate between symbols as it relies on slower, sequential cognitive processes.

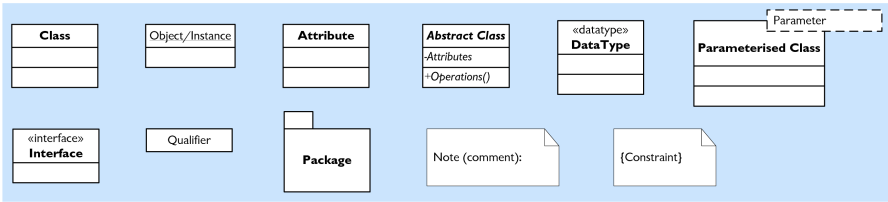















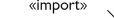
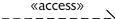
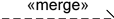

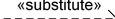
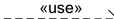
Fig. 6. Class Diagram node types: visual distances between symbols are too small to be reliably perceptible

Table 1 shows the link types most commonly used on Class diagrams. There are a number of discriminability problems with these conventions:

- Visual proximity: the link types differ on only two visual variables (*shape* and *brightness*) and the differences in values chosen are very small in many cases (e.g. closed and open arrows are visually similar).

- Non-unique values: discrimination between relationship types mostly relies on unique combinations of values (conjunctions). Only two of the links have unique values on any visual variable which precludes perceptual popout in most cases.
- Textual encoding: some of the links have zero visual distance (*homographs*) and are differentiated only by labels.

Table 1. Relationship Types in UML Class Diagrams

Aggregation	Association (navigable)	Association (non-navigable)	Association class relationship	Composition
				
Constraint	Dependency	Generalisation	Generalisation set	Interface
				
N-ary association	Note reference	Package containment	Package containment	Package import (public)
				
Package import (private)	Package merge	Realisation	Substitution	Usage
				

Recommendations for Improvement

There is a range of ways to improve the discriminability of UML symbols:

- Clearly distinguishable shapes: of all the visual variables, shape plays a privileged role in discrimination as it is the primary basis on which we classify objects in the real world: differences in shape generally signal differences in *kind* [2, 17, 30]. This means that shapes used to represent different constructs should be clearly distinguishable from each other. For example, packages could be shown as 3D rectangles to differentiate them from classes (Figure 7): these are clearly distinguishable from each other as they belong to different shape families. Another advantage is that their 3D appearance suggests that they can “contain” other things, which provides a clue to their meaning (*Principle of Perceptual Immediacy*).
- Perceptual popout: each symbol should have a unique value on at least one visual variable rather than relying on conjunctions of values.
- Redundant coding: discriminability can also be improved by using multiple visual variables in combination to increase visual distance [16]. Redundancy is an important technique used in communication theory to reduce errors and counteract noise [10, 33]. Figure 7 is an example of the use of redundant coding, as brightness (through the use of shading) is used in addition to shape to distinguish between classes and packages. This also facilitates perceptual popout as no other UML symbol uses shading.

- Resolving symbol redundancy: this principle also provides the basis for choosing between synographs. For example, in Use Case Diagrams, stick figures should be preferred to rectangles for representing actors because they are much more discriminable from all other symbols that can appear on such diagrams.

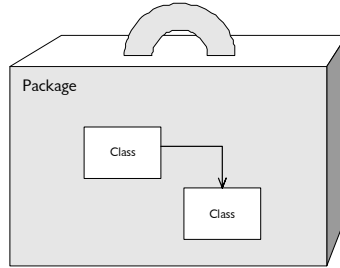


Fig. 7. Use of a 3D shape to differentiate packages from classes. A handle can also be included to reinforce the role of a package as a “container” (Principle of Perceptual Immediacy).

Redundant coding can also be used to improve discriminability of relationships in UML. In the current UML notation (Figure 8: left), the visual distance between generalisation relationships and associations is very small, even though these represent fundamentally different types of relationships (relational vs set-theoretic [12]). Discriminability of these relationships can be improved by using an additional visual variable, *vertical location* (y), to differentiate between them (Figure 8: right). Location is one of the most cognitively effective visual variables and this spatial configuration of elements is naturally associated with hierarchical relationships, which supports ease and accuracy of interpretation (*Principle of Perceptual Immediacy*) [43].

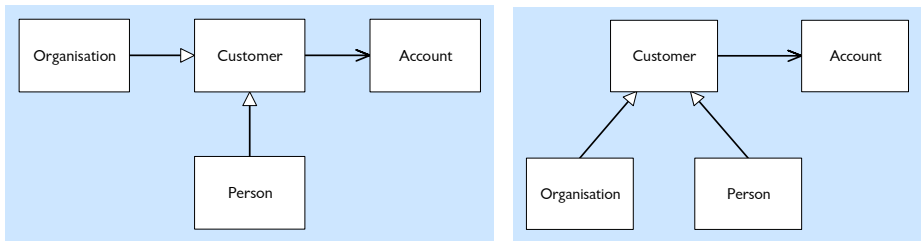


Fig. 8. Redundant coding I. Left: UML 2.0 uses only one visual variable (shape of arrowheads) to distinguish between association and generalisation relationships. Right: vertical location could be used to further differentiate generalisation relationships by always placing subclasses below superclasses.

Discriminability can be improved further by adding a third visual variable: line topology (*shape*) (Figure 9). Merged or “tree-style” line structures (where multiple lines converge into one) are also naturally associated with hierarchical relationships as they are the standard way of representing organisational charts (*Principle of Perceptual*

Immediacy). This would be even more effective if merged lines were reserved for this purpose: currently in UML, merged lines can be used to indicate generalisation sets but can also be used for many other types of relationships (e.g. aggregation, package composition, realisation) which means that it lacks any specific meaning.

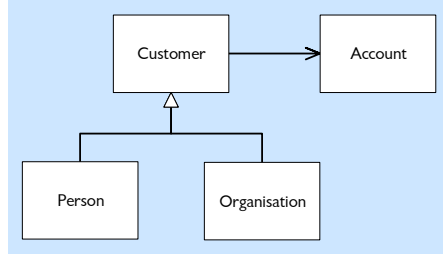


Fig. 9. Redundant coding II. Line topology (*shape*) can be used to differentiate generalisation relationships further

Discriminability can be improved further by adding a fourth visual variable: line weight (*size*) (Figure 10). Use of an ordinal variable such as size creates a natural perceptual precedence among relationship types: generalisation relationships will be perceived to be more important than other relationship types. However this is justifiable as generalisation relationships are arguably the strongest relationships of all (as they represent “family” relationships): their visual precedence is therefore congruent with their semantic precedence.

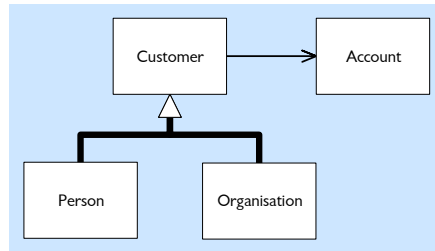


Fig. 10. Redundant coding III. Line weight (*size*) can be used to differentiate generalisation relationships further

Finally, discriminability can be improved even further by adding a fifth visual variable: colour (Figure 11).

Each step in the process progressively introduces a new visual variable, so that by Figure 11 the difference between the relationships is unmistakable. The last three visual variables introduced represent unique values as no other UML relationship types use these values: which facilitates perceptual “popout”. If merged lines were reserved for generalisation hierarchies, this too would be a unique value.

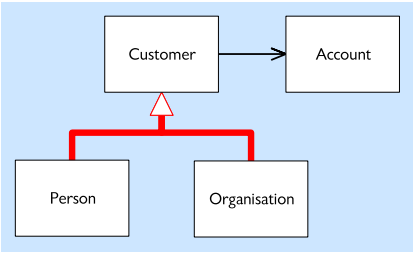


Fig. 11. Redundant Coding IV. Colour can also be used to redundantly encode generalisation relationships

4.3 Principle of Perceptual Immediacy

Like most software engineering notations, UML mostly relies on abstract geometrical shapes to represent constructs. Such symbols don’t convey anything about the meaning of their referent concepts: their meaning is purely conventional and must be learnt. The only exception is Use Case Diagrams where stick figures and custom icons may be used to represent different types of actors. This may be one reason why these diagrams are more effective for communicating with business stakeholders than most other UML diagrams. UML also makes limited use of spatial relationships and relies mainly on different types of connecting lines to distinguish between different types of relationships (e.g. Table 1).

Recommendations for Improvement

Where possible, perceptually direct shapes, icons and spatial relationships should be used instead of simple “boxes and lines”. For example, Figure 7 shows how a more perceptually direct shape could be used to represent packages. As another example, *whole-part relationships* are shown in a similar way to other associations and are differentiated only by a different shaped connector. This convention is not mnemonic (it is difficult for non-experts to remember which end is the “whole” and which is the “part” end) and is ambiguous because diamonds are also used to represent n-ary relationships. A more perceptually direct way to show such relationships would be to allow them to “pierce” elements (Figure 12). This makes such relationships stand out from other relationships (*Principle of Perceptual Discriminability*) and also has a mnemonic association as the part emerges from “inside” the whole. Alternatively, a “dovetail joint” could be used.

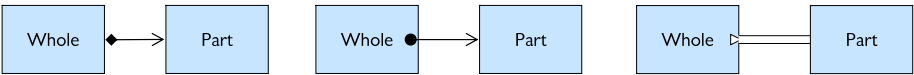


Fig. 12. “Piercing” elements (middle) conveys the concept of being “part of” in a more perceptually direct way than the existing convention (left) and makes such relationships more discriminable from other types of relationships. A “dovetail joint” (right) provides another possible option.

4.4 Principle of Visual Expressiveness

The visual expressiveness of the UML visual notations is summarised in Table 2. Most diagram types use only two of the eight available visual variables, mostly shape and brightness. Shape is one of the least efficient variables and brightness is problematic for discriminability purposes (for which it is mainly used) because it is an ordinal variable so creates perceptual precedence among symbols.

Activity Diagrams are the only diagrams that use more than two visual variables, primarily because they use both planar variables (x,y) to encode information: “swim-lanes” allow the diagram to be divided into horizontal and/or vertical regions which can convey information about who performs the activity, where it is performed etc. The planar variables are the most powerful visual variables and represent the major distinguishing feature between diagrams and textual representations [1, 14]. However few software engineering notations use spatial location to convey information and it acts as a major source of noise on most software engineering diagrams. UML Activity Diagrams represent a rare example of excellence in using spatial location to encode information in the software engineering field.

Colour is one of the most cognitively effective visual variables, yet is specifically avoided in UML:

“UML avoids the use of graphic markers, such as colour, that present challenges for certain persons (the colour blind) and for important kinds of equipment (such as printers, copiers, and fax machines).” [25]

A major contributing factor to the visual inexpressiveness of the UML visual notations is the use of text to encode information. UML currently relies far too much on textual elements to distinguish between symbols and to encode semantics (e.g. cardinalities of relationships). As a result, UML diagrams are really more textual than graphical.

Table 2. Visual Expressiveness of UML Diagrams

Diagram Type	X	Y	Size	Brightness	Colour	Shape	Texture	Orientation
Activity	●	●		●	Specifically prohibited	●		
Class				●		●		
Communication				●		●		
Component				●		●		
Composite structure				●		●		
Deployment				●		●		
Interaction overview				●		●		
Object				●		●		
Package				●		●		
Sequence	●					●		
State machine				●		●		
Timing	●	●						
Use case	●					●		

Recommendations for Improvement

The recommendations for improving visual expressiveness are:

- Colour: this provides an obvious opportunity for improving cognitive effectiveness of UML diagrams as colour is one of the most cognitively efficient visual variables. Differences in colour are detected faster and more accurately than any other visual variable [37, 43]. The reasons given for not using colour in UML are simply not valid: firstly, technology limitations should not drive notation design choices; secondly, graphic designers have known about such problems for centuries and have developed effective ways of making graphic designs robust to such variations (primarily through redundant coding).
- Graphical rather than textual encoding: text is less cognitively effective for encoding information and should only be used as a “tool of last resort” [24]. The more work that can be done by the visual variables, the greater the role of perceptual processes and computational offloading.

4.5 Principle of Graphic Parsimony (Less Is More)

It is not an easy task to determine the graphic complexity of the UML diagram types as the rules for including symbols on each diagram type are very loosely defined. For example, there are 6 different types of *structure diagrams* (Class, Component, Composite Structure, Deployment, Object, Package), but symbols from any of these diagrams can appear on any other (which explains their very high levels of graphic complexity in Table 3 below). This means that the graphic complexity of each type of structure diagram is the *union* of the symbols defined across all diagram types. The reason for such flexibility is hard to fathom, as the point of dividing the language into diagram types should be to partition complexity and to make the visual vocabularies of each diagram type manageable. Having such a *laissez-faire* approach places a great strain on perceptual discriminability and increases the graphic complexity of all diagrams. Table 3 summarises the graphic complexity of the UML diagram types (the

Table 3. Graphic Complexity of UML Diagrams

Diagram	Graphic Complexity
Activity	31
Class	60
Communication	8
Component	60
Composite structure	60
Deployment	60
Interaction overview	31
Object	60
Package	60
Sequence	16
State machine	20
Timing	9
Use case	9

number of different graphical conventions that can appear on each diagram type). Most of them exceed human discrimination ability by quite a large margin. A notable exception is Use Case Diagrams which may explain their effectiveness in communicating with business stakeholders.

Recommendations for Improvement

There are three general strategies for dealing with excessive graphic complexity:

1. Reducing semantic complexity: reducing the number of semantic constructs is an obvious way of reducing graphic complexity. In UML, a major contributor to graphic complexity is the level of overlap between the visual vocabularies of diagram types. To reduce this, metamodel constructs could be repartitioned so that the number of constructs in each diagram type is manageable and there is less overlap between them.
2. Reducing graphic complexity: graphic complexity can also be reduced directly (without affecting semantics) by introducing *symbol deficit*.
3. Increasing visual expressiveness: The span of absolute judgement and therefore the limits of graphic complexity can be expanded by increasing the number of perceptual dimensions on which visual stimuli differ. Using multiple visual variables to differentiate between symbols (*Principle of Visual Expressiveness*) can increase the span of absolute judgement in an (almost) additive manner [18].

5 Conclusion

This paper has evaluated the cognitive effectiveness of the 13 UML diagram types using theory and empirical evidence from a wide range of fields. The conclusion from our analysis is that radical surgery is required to the UML visual notations to make them cognitively effective. We have also suggested a number of ways of solving the problems identified, though these represent only the “tip of the iceberg” in the improvements that are possible: space limitations prevent us including more. Of all the diagram types, Use Case Diagrams and Activity Diagrams are the best from a visual representation viewpoint, which may explain why they are the most commonly used for communicating with business stakeholders [6]. Class Diagrams are among the worst – even though they are the most important diagrams – which may explain why they are commonly translated into text for communicating with users [36].

Another goal of this paper has been to draw attention to the importance of visual syntax in the design of UML. Visual syntax is an important determinant of the cognitive effectiveness of software engineering notations, perhaps even more important than their semantics. We believe that the lack of attention to visual aspects of UML is a major oversight and that as a result its visual development is lagging behind its semantic development.

5.1 Theoretical and Practical Significance

Software engineering has developed mature methods for evaluating semantics of software engineering notations (e.g. ontological analysis). However it lacks comparable methods for evaluating visual notations. The theoretical contribution of this paper is to demonstrate a scientific approach to evaluating and improving visual notations,

which provides an alternative to informal approaches that are currently used. This complements formal approaches such as ontological analysis that are used to evaluate semantics of notations.

The practical contribution of this paper is that it provides a theoretically and empirically sound basis for improving the cognitive effectiveness of the UML visual notations. This will improve communication with business stakeholders (which is currently a major weakness) as well as design and problem solving performance. Cognitive effectiveness supports both these purposes as it optimises representations for processing by the human mind.

5.2 UML 3.0: A Golden Opportunity?

The next release of UML represents a golden opportunity to redesign the visual notations in a cognitively optimal manner. The existing UML notations have been developed in a bottom up manner, by reusing and synthesising existing notations. However ideally, visual representations should be designed in a *top-down manner* based on a thorough analysis of the information content to be conveyed: form should follow content [31]. The UML metamodel provides a sound and now relatively stable foundation for doing this. Of course, such change will need to be handled carefully as practitioners familiar with the existing notation will resist radical change. However they may be more open to changes if they have a clear design rationale grounded on theory and empirical evidence, something which is currently lacking from the UML visual syntax.

References

1. Bertin, J.: *Semiology of Graphics: Diagrams, Networks, Maps*. University of Wisconsin Press, Madison (1983)
2. Biederman, I.: *Recognition-by-Components: A Theory of Human Image Understanding*. *Psychological Review* 94(2), 115–147 (1987)
3. Britton, C., Jones, S.: *The Untrained Eye: How Languages for Software Specification Support Understanding by Untrained Users*. *Human Computer Interaction* 14, 191–244 (1999)
4. Britton, C., Jones, S., Kutar, M., Loomes, M., Robinson, B.: *Evaluating the intelligibility of diagrammatic languages used in the specification of software*. In: Anderson, M., Cheng, P., Haarslev, V. (eds.) *Diagrams 2000*. LNCS, vol. 1889, pp. 376–391. Springer, Heidelberg (2000)
5. Cheng, P.C.-H., Lowe, R.K., Scaife, M.: *Cognitive Science Approaches To Understanding Diagrammatic Representations*. *Artificial Intelligence Review* 15(1/2), 79–94 (2001)
6. Dobing, B., Parsons, J.: *How UML is Used*. *Communications of the ACM* 49(5), 109–114 (2006)
7. Gehlert, A., Esswein, W.: *Towards a Formal Research Framework for Ontological Analyses*. *Advanced Engineering Informatics* 21, 119–131 (2007)
8. Glinz, M., Berner, S., Joos, S.: *Object-oriented modeling with ADORA*. *Information Systems* 27, 425–444 (2002)
9. Goodman, N.: *Languages of Art: An Approach to a Theory of Symbols*. Bobbs-Merrill Co., Indianapolis (1968)

10. Green, D.M., Swets, J.A.: *Signal Detection Theory and Psychophysics*. Wiley, New York (1966)
11. Gurr, C.A.: Effective Diagrammatic Communication: Syntactic, Semantic and Pragmatic Issues. *Journal of Visual Languages and Computing* 10, 317–342 (1999)
12. Harel, D.: On Visual Formalisms. *Communications of the ACM* 31(5), 514–530 (1988)
13. Jonkers, H., van Buuren, R., Hoppenbrouwers, S., Lankhorst, M., Veldhuijzen van Zanten, G.: *ArchiMate Language Reference Manual (Version 4.1)*. Archimate Consortium, 73 (2007)
14. Larkin, J.H., Simon, H.A.: Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognitive Science* 11(1) (1987)
15. Lee, J.: Design Rationale Systems: Understanding the Issues. *IEEE Expert* 12(3), 78–85 (1997)
16. Lohse, G.L.: The Role of Working Memory in Graphical Information Processing. *Behaviour and Information Technology* 16(6), 297–308 (1997)
17. Marr, D.C.: *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. W.H. Freeman and Company, New York (1982)
18. Miller, G.A.: The Magical Number Seven, Plus Or Minus Two: Some Limits On Our Capacity For Processing Information. *The Psychological Review* 63, 81–97 (1956)
19. Moody, D.L.: Evidence-based Notation Design: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering* (under review, 2009)
20. Moody, D.L.: Review of ArchiMate: The Road to International Standardisation, Report commissioned by the ArchiMate Foundation and BiZZDesign B.V., Enschede, The Netherlands, 77 pages (2007)
21. Moody, D.L., Mueller, R.M., Amrit, C.: Review of ORES Methodology, Notation and Toolset, Report commissioned by Egon-Sparenberg B.V., Amsterdam, The Netherlands, 53 pages (2007)
22. Niemela, M., Saarinen, J.: Visual Search for Grouped versus Ungrouped Icons in a Computer Interface. *Human Factors* 42(4), 630–635 (2000)
23. Nordbotten, J.C., Crosby, M.E.: The Effect of Graphic Style on Data Model Interpretation. *Information Systems Journal* 9(2), 139–156 (1999)
24. Oberlander, J.: Grice for Graphics: Pragmatic Implicature in Network Diagrams. *Information Design Journal* 8(2), 163–179 (1996)
25. OMG Unified Modeling Language Version 2.0: Superstructure. Object Management Group, OMG (2005)
26. Opdahl, A.L., Henderson-Sellers, B.: Ontological Evaluation of the UML Using the Bunge-Wand-Weber Model. *Software and Systems Modelling* 1(1), 43–67 (2002)
27. Peirce, C.S.: *Charles S. Peirce: The Essential Writings (Great Books in Philosophy)*. Prometheus Books, Amherst (1998)
28. Petre, M.: Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications of the ACM* 38(6), 33–44 (1995)
29. Quinlan, P.T.: Visual Feature Integration Theory: Past, Present and Future. *Psychological Bulletin* 129(5), 643–673 (2003)
30. Rossion, B., Pourtois, G.: Revisiting Snodgrass and Vanderwart's object pictorial set: The role of surface detail in basic-level object recognition. *Perception* 33, 217–236 (2004)
31. Rubin, E.: *Synsoplerede Figuren*. Gyldendalske, Copenhagen (1915)
32. Shanks, G.G., Tansley, E., Weber, R.A.: Using Ontology to Validate Conceptual Models. *Communications of the ACM* 46(10), 85–89 (2003)

33. Shannon, C.E., Weaver, W.: *The Mathematical Theory of Communication*. University of Illinois Press, Urbana (1963)
34. Siau, K.: Informational and Computational Equivalence in Comparing Information Modeling Methods. *Journal of Database Management* 15(1), 73–86 (2004)
35. Siau, K., Cao, Q.: Unified Modeling Language: A Complexity Analysis. *Journal of Database Management* 12(1), 26–34 (2001)
36. Tasker, D.: Worth 1,000 Words? Ha! *Business Rules Journal* 3(11) (2002)
37. Treisman, A.: Perceptual Grouping and Attention in Visual Search for Features and for Objects. *Journal of Experimental Psychology: Human Perception and Performance* 8, 194–214 (1982)
38. Treisman, A., Gelade, G.A.: A Feature Integration Theory of Attention. *Cognitive Psychology* 12, 97–136 (1980)
39. Treisman, A., Gormican, S.: Feature Analysis in Early Vision: Evidence from Search Asymmetries. *Psychological Review* 95(1), 15–48 (1988)
40. Wheildon, C.: *Type and Layout: Are You Communicating or Just Making Pretty Shapes?* Worsley Press, Hastings (2005)
41. White, A.W.: *The Elements of Graphic Design: Space, Unity, Page Architecture and Type*. Allworth Press, New York (2002)
42. Winn, W.D.: An Account of How Readers Search for Information in Diagrams. *Contemporary Educational Psychology* 18, 162–185 (1993)
43. Winn, W.D.: Encoding and Retrieval of Information in Maps and Diagrams. *IEEE Transactions on Professional Communication* 33(3), 103–107 (1990)
44. Yeh, M., Wickens, C.D.: Attention Filtering in the Design of Electronic Map Displays: A Comparison of Colour Coding, Intensity Coding and Decluttering Techniques. *Human Factors* 43(4), 543–562 (2001)

Neon: A Library for Language Usage Analysis

Jurriaan Hage and Peter van Keeken

Department of Information and Computing Sciences, Universiteit Utrecht
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
Tel.: +31 (30)253 3283; fax.: +31 (30) 251 3791
`jur@cs.uu.nl`

Abstract. A language is ultimately developed for its users: the programmers. The gap between the language experts who design and implement the language and its associated tools and the programmers can be very large, particularly in areas where the language is developed to be used by non-programmers or novice programmers by design. To verify that assumptions made and beliefs held by the language developers about the actual use made by programmers bear out, some analysis of the usage of the language and its associated tools must be performed, preferably in an automated fashion transparent to the users. In this paper we detail our approach, which is essentially to log compilations in order to obtain compilation histories and to query these compilation histories in order to extract the necessary information. For the latter task, we have designed and implemented the Neon library, and show how relatively complicated queries can be formulated effectively.

Keywords: tools for language usage analysis, compilation loggings, system description, functional programming.

1 Introduction and Motivation

A programming language is ultimately developed for its end users: the *programmers*. The gap between the (language) *developers* who design and implement the language and its associated tools and the programmers can be very large, particularly in areas where the language is developed to be used by non-programmers or novice programmers by design. To verify that assumptions made and beliefs held by the developers about the actual use made by programmers bear out, some analysis of the usage of the language and its associated tools must be performed. In our view, such an investigation should be performed in an automated fashion, transparent to the programmers.

In this paper we describe our experiences in creating an environment that allows to investigate the usage properties of a language and its associated tools, by logging each use.

For example, our Helium compiler (which implements a large subset of the Haskell 98 language [13]) may terminate either in one of a number of compiler error phases, or it may terminate due to an internal error (of the compiler), or it is a successful compilation and code is generated (the CodeGen phase). The

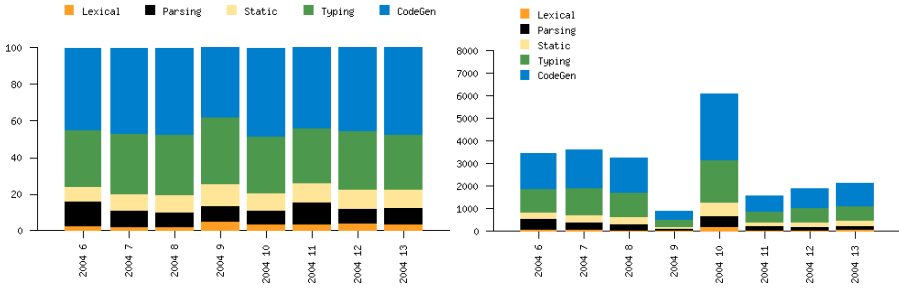


Fig. 1. Relative and absolute number of compiles per phase, given per week from 2003/2004

four most interesting compiler error phases are Lexical, Parsing, Static (simple static errors such as undefined or multiply defined identifiers) and Typing (for a type error). Now we may hypothesize that as students get more experienced in programming Haskell, the ratio of successful compiles increases.

To find support for such a claim, we calculate the ratio between the number of loggings per phase and the total number of loggings in each week. The result of running it on loggings collected from week 6 up to week 13 in 2004 can be found in Figure 1(left).

So how should this figure be interpreted? A first “trend” that may be observed is that the two programming assignments show a similar picture: the ratio of correct compiles gradually increases. But is that due to the fact that programmers are getting more used to the language, or to the assignment? Or is it because they work more on writing comments and prettying their code during the final stages that they make fewer mistakes?

There is more. For example, week 9 seems to paint quite a significantly different picture from the other weeks, so why might that be? Might it be because students work on something else in that particular week? Might it be that a different group of students is at work in the labs? Any of these might be the case. In fact, a possible clarification of the phenomenon might be that the number of loggings for week 9 is much below that of most other weeks, so the ratio computed for the week is not significant. Figure 1(right) which show the absolute number of compiles per phase, seems to indicate as much.

Another somewhat peculiar phenomenon is that the ratio of parse errors decreases, until week 11 when it suddenly strongly increases again. Might this be due to newly introduced language constructs, the start of a new assignment, the good students choosing to work at home, or a combination of all of these. To gain more insight many more queries need to be formulated, executed and interpreted.

This is exactly why Neon encourages reuse by providing general primitives and combinators for quickly building new analyses from others. Moreover, it provides substantial facilities for presenting the results of queries in graphic form. To be productive, we found it necessary that the details of generating graphs and the like should be handled by the library as much as possible. Otherwise

too much time will be spent in getting these tedious details right. In this way, Neon allows us to gradually feel our way across a particular area of investigation until we have evidence from a substantial number of queries to bear out our conclusions. As shown in Section 5, specifying the analysis to generate the picture on Figure 1(right) takes only a small number of lines of (straightforward) code.

As the discussion on how to interpret the pictures generated with Neon indicates, performing a full in-depth study of even a relatively small example of language usage analysis is more than space allows. Therefore we have chosen instead to provide examples that address a broad range of queries, executed on a substantial set of loggings obtained from a first year functional programming course. The purely language usage related queries include the phase query just described, a straightforward query to show how the average module size grows over the course period, a more precise variant where we distinguish code lines from empty lines and comment lines, and finally one to compute how much time it takes to “repair” a type error. We also consider a query that computes the average in-between compilation time for the student population during the course period, an example of a query that has aspects of both language usage and tool usage. Finally, we have included a pure tool usage query which determines the ratio of type error messages for which our compiler produces an additional hint on how to repair the error.

Being able to make qualitative and quantitative statements about the usage of the language and its tools is very important. In this way we can discover, e.g., which parts of the language are used often or avoided, in which situation most of the mistakes are made, and how language usage varies over time as programmers gain experience. With some background information about the programmers, we can investigate other effects on language usage, such as language interference and synergy, due to familiarity with other programming languages. Partly because of these many different purposes for analysing loggings, we have chosen to base Neon on the general concepts of descriptive statistics.

The paper proceeds as follows. After discussing the concrete set-up of our system (Section 2), we consider the concepts behind and the design of Neon in Section 3, followed in Section 4 by a number of sample pictures generated with Neon. In Section 5 we take the reader through a couple of Neon queries, mostly to show the effort involved in writing such queries. For completeness, Section 6 describes in more detail the combinators and primitives supplied by Neon. In Section 7 we discuss related work in the field of language analysis. Section 8 concludes.

2 The Environment

The Helium compiler implements a large subset of the higher-order lazy functional programming language Haskell. Our programmers are first year students. We instrumented the Helium compiler [7] to transmit (log) the source of every compiled module to a simple Java server application. The implementation is

based on sockets and, on both sides, straightforward. Between 2002 and 2006, about 68,000 complete compiles have been logged.

Quite a bit of information is logged: the login name of the programmer who made compilation, the module being compiled, all the modules that it imports, the parameters passed to the compiler, the version number of the compiler and a special code to indicate whether the compilation succeeded and, if it failed, in which phase it failed. Each logging is time-stamped. Most of the logged information is needed to reconstruct exactly the original compilation. Although not necessary for reconstructing the compilation, the time stamp and the login name of the programmer can be very important: you need both to determine, e.g., how much time lies between compiles during a programming session (see Figure 4).

To be more precise, you only need to know which compiles were made by the same programmer, not the identity of the programmer. This is why we have anonymized our loggings in a way that the identities themselves are lost, but loggings by the same programmer are kept together. This may seem in contrast with our statement in the introduction, that background information about the programmer can be important in some studies. The point is that we may retain some information about the programmer, e.g., his number of years of experience with the language, or a list of languages he is familiar with, but we do not retain his exact identity. Obviously, some care must be taken here that the combined background information does not allow us to reconstruct the identity of the programmer. The main goal of anonymization is to prevent personal information leaking through in the experimental results.

We note that loggings can be used for other purposes as well. In our particular situation, internal errors are also logged, and flagged. This allows quick access to problematic programs, and programmers do not have to send their source code manually. The loggings of successfully compiled programs can be used to test and validate optimizations implemented in the compiler, the unsuccessful ones to test and validate improvements in error diagnosis. Both kinds can be used for regression testing: if we, say, refactor the parser, we can apply the new compiler to the logged programs to see if there are any noticeable changes in its responses.

For more details about what is logged exactly, the implementations, ethical considerations, anonymization and other complications, see a technical report on the subject [4].

3 Concepts and Design

The requirements. The main requirements we set for Neon were the following: (1) Queries should be concise and conceptually close to what they intend to express. (2) Its implementation should be based on a small set of well-understood primitives and combinators, to make it easier to argue that the implementation of Neon is correct. (3) It should be easy to reuse code from the compiler, since we expect to need the lexer, parser and other parts that make up the compiler for some of our queries. (4) Finally, Neon should generate esthetically pleasing

output, supporting multiple output formats, e.g., HTML tables and PNG files. This greatly simplifies reporting on the results of the analyses.

The next two sections mean to show that we fulfill the first two of these requirements. We consider the third and fourth requirement in this section.

Meeting the third requirement is easy enough: Neon is implemented in Haskell as is Helium. Although not an essential requirement, it does simplify some things: consider a query to determine the maximum number of parameters for any definition in the program. We can of course implement this in any general programming language, but it makes a lot more sense to reuse the parser that is part of Helium, both to save effort and to guarantee consistency. In that case, a simple traversal of the abstract syntax is all we need to write. The gains become larger the more we can reuse. If, for example, we want statistical information about the sizes and number of dictionaries passed to deal with type classes [15], then this can only be computed after type inferencing has taken place. Again something we prefer not to reimplement.

This does not yet rule out implementing Neon in another language. For our experimental results of [5] for example, we instrumented the Helium compiler to emit information about the contribution that particular type heuristics made to a type error message. We then used Perl to collect the emitted information and summarize it. As a quick hack this actually works quite well, but there are problems too. First of all, for each query of this type we need to (further) instrument the compiler and provide compiler flags to tell it to actually output this information. To avoid this pollution of the compiler, we prefer to keep the compiler and Neon separate, and maximize reuse in order to save effort. This is one of the reasons why we implemented Neon in Haskell. But it is not the only one.

When we started out it was not yet clear what the right primitives and combinators were going to be. One of the nice properties of Haskell is that it easily allows us to grow a domain specific language (DSL) as a library inside the language. This also avoids having to build a compiler for the DSL. The main downside of this approach is that error messages suffer, because the Haskell compiler is not “DSL aware” and error messages are phrased in terms of the host language (but see [6] for a solution to this problem). In our case, the problem is not so pressing, because the types in Neon are not very complicated.

Combinators can be used to construct analyses out of others. Since analyses are essentially functions, combinators are often functions that take a function to deliver a function; here the higher-orderness of Haskell is essential to keep the code clean. Haskell’s parametric polymorphism allows us to easily abstract away from the actual information that is being analysed. Finally, the type classes idiom [15] allows to hide many tedious details, e.g., for generating pictures and for transforming keys (see Section 5), inside a type class. This simplifies the use of the analysis functions, while the general underlying functions are still available when the need arises.

A natural question to ask at this point is to what extent other languages besides Haskell can profit from using Neon. The use of Neon is, in principle, not tied to any particular language, and in fact, most of the queries we consider

below do not consider the contents of the logged sources and thus do not depend on the fact that the source files contain Haskell code.

As we explained above, queries that are intrinsically tied to the syntax and static semantics of the programming language under analysis may benefit from the fact that Neon and the compiler are written in the same language, because we can then use parts of the compiler directly in Neon. In a situation where this is not the case, there are several possibilities:

- to reverse engineer parts of the compiler, implement these in Haskell and use them from Neon,
- to instrument the compiler (as in the Perl example), or
- to reverse engineer Neon in the language the compiler was built in.

The best choice obviously depends very much on the context. If the third alternative is chosen, then the implementation of Neon, the concepts behind it and our experiences with the tool can still serve as a source of inspiration.

The fourth requirement is met by mapping our analysis results to a number of output formats, including a file format that is used by the `ploticus` program [2] for generating pretty pictures (see Section 4). The library provides various functions to generate pictures and tables. Haskell’s type classes are used to further simplify the generation of pictures, for example by making sure that the same values occur in fixed order, e.g., if values are of a nominal kind and do not have a natural order associated with them. This facility is very useful if you want to apply the same analysis to several subsets of the compilation loggings, and want to be able to compare the outcomes easily.

The language concepts. Instead of reinventing the wheel, we have taken the concepts on which to base the library from the area of descriptive statistics, which deals with how to summarize (large amounts of) data¹. Specifically, there are provisions for dealing with the following issues:

- to group loggings (repeatedly) into groups of related loggings,
- to compute statistical or computational characteristics of the loggings in each group,
- to select individual loggings or groups of them based on some computed characteristic, and
- to present the results in tables and pictures.

Currently, the logged sources are stored in the file system. Since the files are typically small and similar this can certainly be improved upon whenever storage and speed of access become an issue.

We note that interfacing with a database management system is an option, but, for now, we kept the number of dependencies on other libraries as small as possible. The only dependency we currently have is on the `ploticus` program [2] for generating the pictures.

The Neon library is available for download [3]. It includes a small anonymized excerpt from our collected loggings for demonstration purposes and, among others, the code for the examples in this paper.

¹ http://en.wikipedia.org/wiki/Descriptive_statistics

4 Example Output

In Section 5 we consider two analyses in detail and show how pictures can be generated with Neon. In this section, we limit ourselves to show a selection of pictures generated with Neon. The code for these examples can be found in the master thesis of the second author [14].

In Figure 2 we start out simple by computing how the length of compiled modules evolves over a course, and give the results per week (left; the values on the x-axis indicate the year and week number) and per weekday (right; only those days on which compiles were made are shown). Counting the number of lines is maybe a bit too crude, so we refine it some by considering empty lines, comment lines and code lines separately. In Figure 3, we display the median sizes for each category per week on the left, and on the right give the relative fractions.

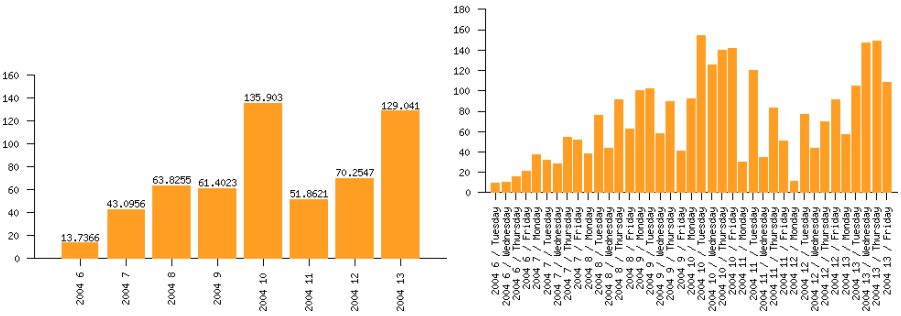


Fig. 2. Average module length in terms of lines per sourcefile per week (left) and per day (right) based on the 2003/2004 data set

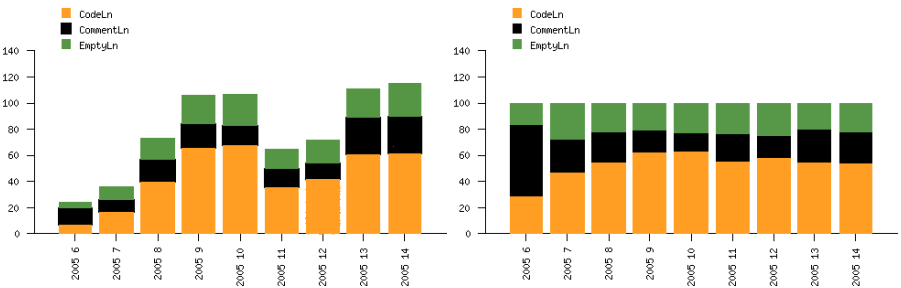


Fig. 3. Median size (left) and relative fractions (right) of source program segments for all students, given per week from 2004/2005

To get an idea how often students compile and how this changes over time, Figure 4 (left) shows a box plot with the averages over the student population, given in minutes for each week. For example, in week 6 of 2004, compiles of

$N = 77$ students were logged. The box indicates the extent of the 25th to the 75th percentile (i.e., it contains the median fifty percent of the values), while the “whiskers” extend from the 5th percentile to the 95th (except that the box is plotted on top of the whiskers). Outliers that lie within (picture) range are plotted as small horizontal lines. The lighter dot in the box indicates the median value, the darker dot indicates the average. Hence, the average for week 6 is about 3.5 minutes, the median slightly lower. Similarly, Figure 4 (right) shows a box plot with the medians over the student population. Median values are often favoured over averages, because they are less sensitive to outliers.

Of course, we do not want to count the time that a student spends in class or in bed as in between compilation time. So the query takes a time-coherence parameter m which specifies that after m minutes without compilation by a particular student, we consider the programming session to have ended. While computing the compilation intervals, we make sure that time between sessions is not counted. In Figure 4 we use $m = 60$.

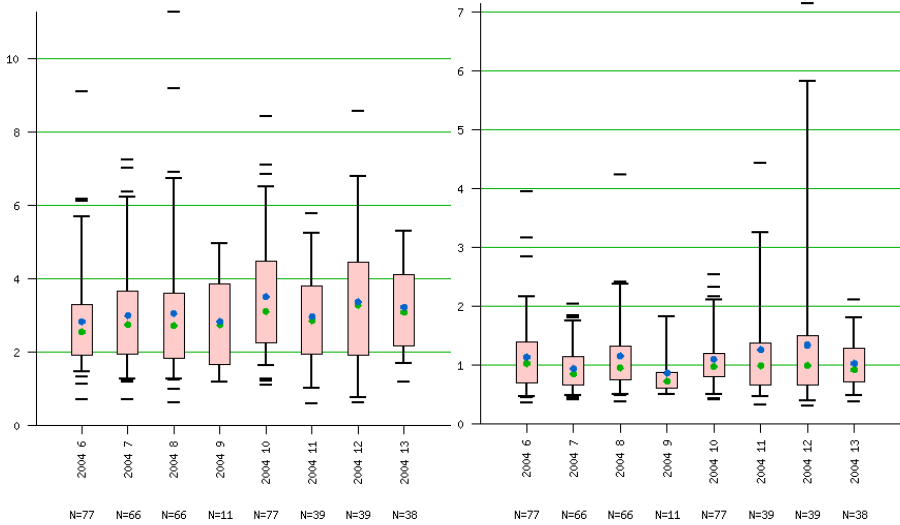


Fig. 4. Average (left) and median (right) compilation intervals (in minutes) for all students, given per week with a time coherence of $m = 60$ minutes from 2003/2004

In Helium, type errors are accompanied by a hint, if one of our heuristics has observed a particular pattern in the error. This hint is supposed to help the programmer correct the mistake. In Figure 5 we show the ratio of type errors that has a hint, both in graphic and table format. We see a strong decrease in this percentage over the course period. There may be various reasons for this. For example, our hints may help the student avoid making the same mistake again, or it may be that the kind of error for which we provide hints are the kind that students tend learn not to make as they get more experienced.

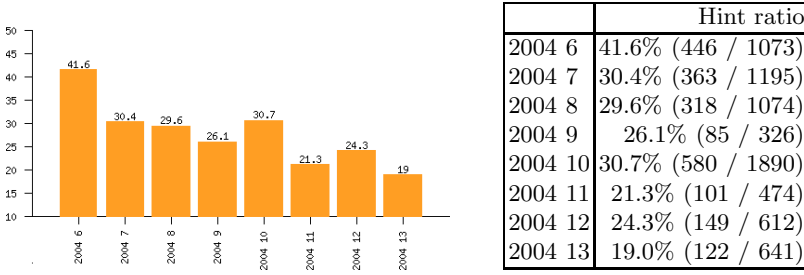


Fig. 5. Hint ratio for type incorrect compilations, from 2003/2004, given per week

5 Analysis Code Examples

In this section, we explain the code of two analyses to illustrate how Neon can be employed. The main purpose is to show how queries can be formulated and to show that they are indeed conceptually close to what they intend to compute. The Haskell code is of itself not very complicated, and we hope that it is intuitive enough for those not (very) familiar with the language. The code is not complete: we omit explanations for some functions, either because they are straightforward or simply not very interesting.

Representation. Before we proceed to the analyses themselves, we first describe how analyses, analysis results and loggings are represented. An analysis result is simply a list of key-value pairs, $[(key, value)]$. Here, *value* is the result of performing the analysis and *key* is a *description* of this value. An analysis simply maps between two types of this kind:

type *Ana keya a keyb b* = $[(keya, a)] \rightarrow [(keyb, b)]$

To be able to compose analyses easily, we have chosen to always map a list of pairs to a list of pairs, even if an operation like grouping is involved. For example, suppose we have the following intermediate result: $[("Bob", ls1), ("Kim", ls2)]$, in which *ls1* and *ls2* contain the loggings of Bob and Kim respectively. Suppose the next operation is to group all the loggings for each week together (per student). If Bob has loggings in week 1 and week 2 and Kim only in week 2, then the result of this operation would be

$[("Bob; week 1", ls11), ("Bob; week 2", ls12), ("Kim; week 2", ls22)]$

and not

$[("Bob", [("week 1", ls11), ("week 2", ls12)]),$
 $(("Kim", [("week 2", ls22)]))]$

Neon is based on a small set of primitives and combinators that can be used to implement analyses of the general type given above. This generality can be useful, but also makes the primitives harder to use: the programmer must describe how both key and analysis values should be transformed. Therefore, Neon provides a special (history) key datatype *KH* that can faithfully describe all the primitive

operations explicitly, and versions of the primitives that use this datatype by default. As a result the structure of the analysis computation will be explicitly represented in the key and the programmer does need to specify the updates to the key values himself.

Moreover, by means of a Haskell type class, *DescrK*, the changes to the key values can be described once and for all in the instance declaration for *KH*. Another reason for using *KH*, is that it can also be used on the presentation side, for example to generate legends and captions. The details of the *DescrK* type class and the *KH* datatype are not particularly important for the Neon programmer, so we omit them here [14]. Because in any transformation the key type will be the same, we typically use a simplified version of *Ana*:

type *AnaF* *key* *a* *b* = *Ana* *key* *a* *key* *b*

Finally then, a *Log* is a record that contains all the general information associated with a particular logging: a *username*, the *heliumVersion*, the *phase* in which the compilation ended, the *logPath* that points to the logged sources, and a time stamp for the logging called *logDate*. In Haskell these fields also serve as the function that can be used to extract the corresponding value from a *Log* value.

In summary, the *Log* datatype describes the contents of our “database”, a value of type *KH* describes which operations (grouping, filtering) have been performed on the logging data, and the *DescrK* type class relieves the programmer from specifying how the *KH* values should be updated.

In our examples, we shall use a small number of primitives, that will be introduced informally along the way. A more complete treatment can be found in Section 6; a quick reference can be found in Figure 6.

5.1 Phase Analysis

The first query to consider in depth is the phase analysis query that was used in the introduction. To find support for the claim that the ratio of correct compiles increases, the analysis calculates the number of loggings per phase, per week, and the ratio between the number of loggings per phase and the total number of loggings in each week. We presented the results as stacked bar charts in Figure 1. We only describe how to compute the absolute numbers, on the right; computing the ratios is then simple.

We first define *groupPerPhase* to group together loggings that ended in the same phase, using the *groupAnalysis* primitive. It uses the auxiliary function *groupAllUnder* :: *Eq* *b* ⇒ (*a* → *b*) → [*a*] → [[*a*]] that takes a function *f* and list *xs* and partitions *xs* into lists, in which two values *x1* and *x2* end up in the same list if and only if *f* *x1* equals *f* *x2*. The duplication of *phase* in the arguments to *groupAnalysis* is due to the fact that the second of these is responsible for actually grouping the arguments, while the first of these handles the automatic update of the *KH* value associated with a sequence of loggings.

groupPerPhase :: *DescrK* *key* ⇒ *AnaF* *key* [*Log*] [*Log*]
groupPerPhase = *groupAnalysis* *phase* (*groupAllUnder* *phase*)

basicAnalysis :: *DescrK key* \Rightarrow *String* \rightarrow (*a* \rightarrow *b*) \rightarrow *AnaF key a b*
 A basic analysis that maps from *a* to *b* and updates the key with a *String*.

groupAnalysis ::
 (*DescrK key*, *DataInfo b*) \Rightarrow (*a* \rightarrow *b*) \rightarrow (*[a]* \rightarrow *[[a]]*) \rightarrow *AnaF key [a] [a]*
 Performs a grouping, always flattens the result.

(*<.>*) :: *Ana kb b kc c* \rightarrow *Ana ka a kb b* \rightarrow *Ana ka a kc c*
 Compose two analyses.

(*<\$>*) :: *Ana keya a keyb b* \rightarrow *[(keya, a)]* \rightarrow *[(keyb, b)]*
 Analysis application, but with lowest possible precedence.

(*\times*) :: *Ana ka a kb1 b1* \rightarrow *Ana ka a kb2 b2* \rightarrow *Ana ka a (kb1, kb2) (b1, b2)*
 Applies two analyses in tandem to a single intermediate result.

splitAnalysis :: *[(key, a)]* \rightarrow *[[(key, a)]]*
 Prepare for running an analysis on all parts of an intermediate result.

mapAnalysis :: *mapAnalysis* :: (*DescrK key*) \Rightarrow *AnaF key a b* \rightarrow *AnaF key [a] [b]*
 Lift an analysis from values to lists.

concatMapAnalysis :: *AnaF KH a [b]* \rightarrow *AnaF KH [a] [b]*
 As *concatMap*, but for analyses.

runAnalysis :: *a* \rightarrow *keya* \rightarrow *Ana keya a keyb b* \rightarrow *[(keyb, b)]*
 Actually run an analysis.

renderXXX
 A family of display functions; choose a suitable *XXX*.

Fig. 6. Quick reference for primitives and combinators

The *loggingsPerPhase* analysis first selects the loggings that are associated with phases in which we are interested, applies *groupPerPhase* and then simply tallies the number of loggings for each phase. Removing loggings that are not interesting is performed by *mainPhasesAnalysis* which is implemented as a basic analysis. It is parameterised by the transformation it should apply, in this case a filter. Note that *<.>* denotes analysis composition. It is similar to function composition.

```

loggingsPerPhase :: AnaF KH [Log] Int
loggingsPerPhase =      countNumberOfLoggings
                      <.> groupPerPhase
                      <.> mainPhasesAnalysis

mainPhasesAnalysis :: AnaF KH [Log] [Log]
mainPhasesAnalysis = basicAnalysis "" (filter (anyfrom mainphases.phase))

where
  mainphases = [Lexical, Parsing, Static, Typing, CodeGen]
  anyfrom    = flip elem

countNumberOfLoggings :: DescrK key  $\Rightarrow$  AnaF key [a] Int
countNumberOfLoggings = basicAnalysis "number of loggings" length

```

Composing *loggingsPerPhase* with *groupPerWeek*, which groups the loggings based on the week numbers obtained from the *logDate* of each logging, yields the information we are after. The function *renderBarChartDynamic* then generates a *ploticus* file [2] which can then be used to generate Figure 1 (right). After computing the ratios, the picture on the left can be generated in a similar fashion.


```

phaseResearch :: FilePath → [(KH, [Log])] → IO ()
phaseResearch dir input = do
  renderBarChartDynamic dir ((loggingsPerPhase <.> groupPerWeek)input)
  return ()

```

5.2 Type Error Repair Analysis

Type errors are the most commonly occurring errors as shown by the phase analysis in Figure 1. As a more complicated example, we investigate how long students take to repair a type incorrect program. We may expect that due to experience, this decreases over time. On the other hand, we also expect that programs become more complex, which may result in type errors that are harder to repair. We want to address the following claim.

Claim: *The time (expressed in seconds spent) needed to repair a type error, decreases over time.*

Analysis design. First, we must specify how we can detect the process of correcting a type incorrect program within the logging sequence of a particular student. We take a simple approach in which we consider the first type error, and consider the sequence of loggings up to the first successful compilation, a so-called *type correcting sequence*, and we repeat. For example, denoting a lexical error, parse error, type error and successful compile with L, P, T and C respectively, we obtain $[T\ T\ T\ P\ T\ C]$ and $[T\ C]$ from $[C\ P\ T\ T\ T\ P\ T\ C\ L\ L\ P\ P\ T\ C\ P]$ (in the code, the function *calcTCSequences* performs this task).

There are two important issues here that should not be forgotten: the compiles should all deal with the same program (approximated by only looking at subsequent compiles of modules with the same name), and we want to avoid overly long in-between compile times: if a student makes a type error on Tuesday, and correctly compiles for the first time on Thursday, then we do not want to take that period into account. Summarizing, subsequent compiles in a type correcting sequences should be coherent in time and content: they are not spaced too far apart (a parameter to the function that computes the coherent sequences) and they should refer to the same module.

The remainder is quite straightforward: for each student, divide the complete sequence of loggings into type correcting sequences and compute the differences in time between the first and last compile in each sequence. The auxiliary function *calcTimeAndFilenameCoherentSeqs* breaks up the logging sequence into subsequences that deal with the same source filename and in which subsequent loggings are no more than a certain number of minutes apart. It is based on the use of the general *groupByCoherence* function that uses two simple functions to turn a list of loggings into a list of lists of loggings: *sameSourceFile* determines whether two loggings deal with the same filename, and *logTimeDiff* which determines the time between two loggings. The functions *minutesOfTD* and *timeDiffToSecanalysis* are simple time conversion functions.

```

calcTimeAndFilenameCoherentSeqs :: Int → AnaF KH [Log] [[Log]]
calcTimeAndFilenameCoherentSeqs min =
  basicAnalysis ("coh.: filename and time (" ++ show min ++ ")")
    (groupByCoherence
      (λl1 l2 → sameSourceFile l1 l2
        ∧
        logTimeDiff l1 l2 < minutesOfTD min
      ))

```

To calculate the time needed to repair a type incorrect program, we construct the *timeTCSequences* analysis. It calculates the difference in time between the first logging and the last logging of a sequence of loggings. The call to *concatMapAnalysis* is responsible for computing the type correcting sequences, as illustrated earlier. The function *timeToCompile* gathers all the ingredients together.

```

timeTCSequences :: AnaF KH [Log] TimeDiff
timeTCSequences = basicAnalysis "" (uncurry logTimeDiff . headlast)
timeToCompile :: Int → [(KH, [Log])] → [(KH, [Integer])]
timeToCompile mintime =
  ( mapAnalysis timeDiffToSecAnalysis
  <.> mapAnalysis timeTCSequences
  <.> concatMapAnalysis (basicAnalysis "" calculateTCSequences)
  <.> calcTimeAndFilenameCoherentSeqs mintime
  <.> groupPerWeek)

```

The analysis just described works on a sequence of loggings for a particular student. To apply the analysis to each student, we split up the loggings into subsequences (one for each student) and then apply the above analysis to each subsequence. This is captured generically by *analysisPerStudent* below to which we should pass *timeToCompile*. In Haskell, function application has highest precedence. In the code below we use both Haskell's *\$* which is an infix operator that also performs function application but with the lowest possible precedence. Our own *<\$>* behaves similarly, but restricted to analyses.

```

analysisPerStudent :: AnaF KH [Log] a → [(KH, [Log])] → [(KH, a)]
analysisPerStudent ana input =
  map ana.splitAnalysis $ groupPerStudent <$> input
timeToCompilePerStudent :: Int → OutputFormat → FilePath
  → [(KH, [Log])] → IO ()
timeToCompilePerStudent mintime format outputpath input = do
  let
    name      = researchname
    researchdir = joinDirAndFileName outputpath (stringToFilePath name)
              ++ [pathSeparator]

```

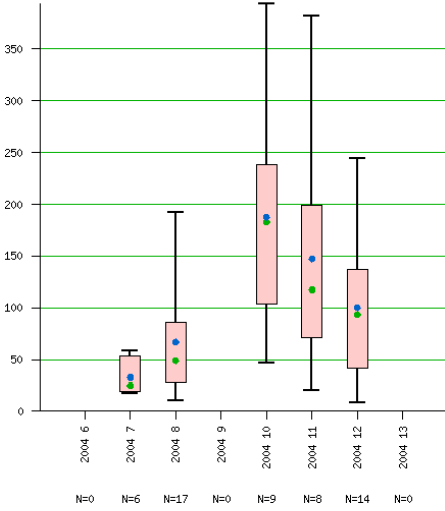


Fig. 7. Time (in seconds, 10 minutes time coherence) needed to repair a type incorrect program for a particular student, from 2003/2004

```

analysis    = timeToCompile mintime
createDirectoryIfMissing True researchdir
mapM_ (renderBoxPlot researchdir) (analysisPerStudent analysis input)

```

The main difference in presenting the results of *analysisPerStudent*, as compared to *phaseResearch*, is that *mapM* maps the presentation function over the results, so we obtain a separate *ploticus* file for each student.

Results. In Figure 7 one of the pictures generated by *timeToCompilePerStudent* is shown.; again the average time is given by the darker dot, the median value by the lighter one. Note that weeks 6 and 9 are included in the picture, although this particular student had no type correcting sequences in those weeks. Neon has a special type class *DataInfo* that handles the insertion of missing values, so that the x-axes of the pictures generated for different students can be standardized and therefore more easily compared.

6 The Neon Internals

This section describes some of the primitives and combinators Neon provides, particular those used in the implementation of the examples found in this paper; more can be found in Chapter 5 and Chapter 6 of the master thesis of the second author [14].

Neon contains basic combinators that abstract away from the particular types of key and value, and as a result their implementation is straightforward; these are the functions that do the actual work. On top of that we have a layer of

functions with similar functionality, but restricted to a particular kind of key, i.e., KH . These are the functions that are to be used by the analysis programmer.

Primitives and Combinators

As pointed out earlier, the primitives derive from the area of descriptive statistics. The combinators, on the other hand, are higher-order functions that construct analyses out of other analyses.

The basic operation of calculating a new value from a previously computed value, can be implemented by the *basicAnalysis* primitive. There are slightly different variants available, but a typical one is the following:

$$\begin{aligned} \text{basicAnalysis} &:: (ka \rightarrow kb) \rightarrow (a \rightarrow b) \rightarrow \text{Ana } ka \ a \ kb \ b \\ \text{basicAnalysis } kf \ vf &= \text{map } (\lambda(k, v) \rightarrow (kf \ k, vf \ v)) \end{aligned}$$

The first function argument specifies how the key values should be updated, while the second actually describes the operation performed on the experimental data.

In the following definition the key type is *String* and the description simply appends a piece of text to describe the operation that is performed, which is to compute the length of a sequence of loggings.

$$\text{countLoggings} = \text{basicAnalysis } (+"; \text{ Number of loggings}") \text{ length}$$

To specify a grouping, we define the *groupAnalysis* combinator that takes a function that determines which values belong together in a group, and a function that actually performs the grouping:

$$\text{groupAnalysis} :: (a \rightarrow k1 \rightarrow k2) \rightarrow ([a] \rightarrow [[a]]) \rightarrow \text{Ana } k1 \ [a] \ k2 \ [a]$$

Implementation is different from *basicAnalysis* since the result of applying the value transformation is a list of lists, which is subsequently flattened before it is returned as the result of the analysis. In this case the key transformation function is a bit more complicated. To be able to compute a value that describes the outcome, we pass an element of the list (in this case the first) and the old key (describing the computations done so far). Usually a grouping collects together the loggings that share a given property, say the week in which a logging was collected. The key transformation function can then obtain that week number of the group from its first argument, and combine it with the old key to obtain the new one.

Analyses can be composed with the $\langle . \rangle$ combinator, which is similar to function composition.

$$\langle . \rangle :: \text{Ana } kb \ b \ kc \ c \rightarrow \text{Ana } ka \ a \ kb \ b \rightarrow \text{Ana } ka \ a \ kc \ c$$

Sometimes, $\langle \$ \rangle$ can be used to simplify the code for an analysis. Like its Haskell equivalent, $\$$, it is an explicit application operator that has lowest precedence among the operators.

The (\times) operator tuples two analyses, in case they are to be applied independently to the same input.

$$(\times) :: \text{Ana } ka \ a \ kb1 \ b1 \rightarrow \text{Ana } ka \ a \ kb2 \ b2 \rightarrow \text{Ana } ka \ a \ (kb1, kb2) \ (b1, b2)$$

It often happens that an aggregate analysis applied to the whole of the loggings shows a trend that needs further investigation. This is typically followed-up by applying the same analysis separately to, e.g., each student. The function *splitAnalysis* can be used here. First group the loggings per student, apply *splitAnalysis*, and then apply the exact same analysis that was applied to the complete set of loggings. The result is a sequence of pictures, one for each student, but the computed property is the same as for the population as whole. Again, *splitAnalysis* has a straightforward implementation:

$$\begin{aligned} \text{splitAnalysis} &:: [(key, a)] \rightarrow [[(key, a)]] \\ \text{splitAnalysis } \text{anareult} &= [[x] \mid x \leftarrow \text{anareult}] \end{aligned}$$

Finally, *mapAnalysis* lifts an analysis from a to b to an analysis from $[a]$ to $[b]$.

$$\begin{aligned} \text{mapAnalysis} &:: \text{Ana } key_a a \ key_b b \rightarrow \text{Ana } key_a [a] \ key_b [b] \\ \text{mapAnalysis } \text{ana} &= \\ &\quad \text{map } (\lambda(key, value) \rightarrow (\text{head } \$ (\text{getKeyTransf } \text{ana}) \ key \\ &\quad \quad \quad , \text{concat } \$ \text{map } (\text{getAnalysisFun } \text{ana}) \ value))) \end{aligned}$$

Specializations of the Primitive Functions

The first step in specialization is the introduction of the *Key* type class. It handles some of the administration of keys, by specifying a start key for a given type of key, and by specifying how keys can be combined into new keys. Implicit in the *Key* class is that the input and output key are of the same type.

type *AnaF* *key* $a \ b = \text{Ana } key \ a \ key \ b$

The most important type class is *DescrK* that encapsulates a fixed (but possibly parameterized) key transformation function for each primitive analysis on the instance data type. By making a type an instance of this type class, special primitives can be used to specify how the key should be transformed. For example, the *groupAnalysis* primitive now becomes:

$$\begin{aligned} \text{groupAnalysis} &:: (\text{DescrK } key, \text{DataInfo } b) \Rightarrow \\ &\quad (a \rightarrow b) \rightarrow ([a] \rightarrow [[a]]) \rightarrow \text{AnaF } key \ [a] \ [a] \end{aligned}$$

The first argument, of type $a \rightarrow b$, describes the property on which grouping takes places, while the second tells us how the grouping should be performed. In

groupPerPhase = *groupAnalysis phase (groupAllUnder phase)*

we want to collect all loggings in the same phase together, whether they are adjacent in the original sequence or not (this is what *groupAllUnder* does). In this particular instance, the type a is *Log* and b is the type *Phase*. The *DataInfo* b constraint is used to automatically deal with missing values: it can make sure that a value for each possible phase is present in the output, even if no compile for a particular phase is present in the input. This is essential if we easily want to generate graphical pictures of this kind over a population of students: in that case we want all the pictures to show exactly the same values

on the x-axis (in this case, the possible phases), and in the same order. This information is captured by the *DataInfo* type class, and is used by presentation functions such as *render1DTableSmart*.

Neon has presentation functions that assume the use of the *KH* datatype to represent keys. Some of these generate textual output, in the form of a *MarkupDoc* that abstracts away from a particular textual representation such as LaTeX or HTML (the table in Figure 3 was generated in this way).

$$\begin{aligned} \text{showAsTable2D} :: (\text{Show } a, \text{Ord } b) \Rightarrow \\ [(KH, [(a, b)])] \rightarrow \text{MarkupDoc} \end{aligned}$$

In addition, presentation functions are provided that turn an analysis result into a *ploticus* file. A *ploticus* file can then be transformed into a graphical format such as PNG, PS or GIF. At this moment, bar charts, stacked bar charts, relative stacked bar charts, dynamic bar charts and box plots are supported. For example, to render a box plot use

$$\begin{aligned} \text{renderBoxPlot} :: (\text{Show}, \text{Num } b, \text{Ord } b) \Rightarrow \\ \text{FilePath} \rightarrow [(KH, [b])] \rightarrow \text{IO } (\text{FilePath}, \text{String}) \end{aligned}$$

7 Related Work

In this section, we limit ourselves to (empirical) studies that have been performed to analyse language usage. As far as we have been able to determine, the investigation of the kind of properties we are interested in, is still very much in its infancy, and this holds for most programming languages. Scouring the Internet we found only a few papers that consider issues related to ours, and relatively scattered throughout time.

Starting in the seventies, a number of studies considered the programming behaviour for various imperative languages: Moulton and Muller [12] evaluated the use of a version of FORTRAN developed especially for education, Zelkowitz [16] traced runs of programs written by students to discover the effects of a structured programming course, and Litecky and Davis [11] considered 1,000 runs from a body of 50 students and classified their mistakes.

A related area of research is the investigation in which way language features influence how easily students learn to program and/or internalize certain aspects of a programming language. These studies usually do not concern themselves with actual feedback provided by programming environments. This type of study goes back to the work of Gannon and Horning [1] in which they compare two syntactically different but similar programming languages. Also in this case, there does not seem to be much recent activity in this field, and our work can surely contribute to this area, e.g., by examining interference or synergy between languages.

Work was done in the early nineties at Universiteit Twente on teaching the functional programming language Miranda to first-year students [10]. The study was performed empirically by following and interviewing a subset of a group of students enrolled in their first-year functional programming course. The outcomes

are of various kinds: they identify problems when learning Miranda, they discovered interferences with a concurrent exposure to an imperative language, and they even went as far to compare problem solving abilities of students who only did the functional programming course with those who did only the imperative programming course. As far as we could tell, there was no automation involved.

A more recent study was performed by Jadud by instrumenting BlueJ (a Java programming environment) to keep track of the compilation behaviour of students [8]. In his study he determines for various types of errors how often they occur. The similarity with our goals is that he also considers compilation behaviour as the subject for analysis. Since imperative languages usually do not have a complex type system, the focus in this work is, like its predecessors from the seventies, on syntactic errors in programming. However, with the addition of parametric polymorphism to Java, and the resulting complication of the type inference process, it is to be expected that type errors will become a fertile area of investigation as well. In a later paper [9], Jadud considers tool support developed for his studies. It consists a browser for viewing source code, a visualization tool that shows the types of parse errors made and their frequency, and an algorithm to score programming sessions. There does not seem to be any support for phrasing and executing other queries besides those provided by the tools.

8 Conclusion

In this paper, we have detailed our approach to language usage analysis. We have described our use of logged compilation to obtain a large collection of programs written by first year functional programming students. In order to query this collection effectively, i.e., with little effort on our part, we have implemented the Neon library based on the concepts of descriptive statistics. With Neon we can quickly create queries to compute information useful to tool builders, language developers, but also people interested in the psychology of programming; we believe that such a tool is essential to effectively perform studies of this kind, especially since we are dealing with thousands of loggings. We have given examples of query results and have taken the reader through a number of example queries. The system is available for download [3]. We invite the reader to give it a try.

Acknowledgments. We thank Alex Gerdes, Bastiaan Heeren, Stefan Holdermans, Johan Jeuring and Michael Stone for their kind support.

References

1. Gannon, J.D., Horning, J.J.: The impact of language design on the production of reliable software. In: Proceedings of the international conference on Reliable software, pp. 10–22. ACM Press, New York (1975)
2. Grubb, S.: Ploticus website, <http://ploticus.sourceforge.net>
3. Hage, J.: Neon website, <http://www.cs.uu.nl/wiki/Hage/Neon>
4. Hage, J.: The Helium logging facility. Technical Report UU-CS-2005-055, Department of Information and Computing Sciences, Utrecht University (2005)

5. Hage, J., Heeren, B.: Heuristics for type error discovery and recovery. In: Horváth, Z., Zsóka, V., Butterfield, A. (eds.) IFL 2006. LNCS, vol. 4449, pp. 199–216. Springer, Heidelberg (2007)
6. Heeren, B., Hage, J., Swierstra, S.D.: Scripting the type inference process. In: Eighth ACM Sigplan International Conference on Functional Programming, pp. 3–13. ACM Press, New York (2003)
7. Heeren, B., Leijen, D., van IJzendoorn, A.: Helium, for learning Haskell. In: ACM Sigplan 2003 Haskell Workshop, pp. 62–71. ACM Press, New York (2003), <http://www.cs.uu.nl/wiki/bin/view/Helium/WebHome>
8. Jadud, M.C.: A first look at novice compilation behaviour using BlueJ. *Computer Science Education* 15(1), 25–40 (2005)
9. Jadud, M.C.: Methods and tools for exploring novice compilation behaviour. In: ICER 2006: Proceedings of the 2006 international workshop on Computing education research, pp. 73–84. ACM Press, New York (2006)
10. Joosten, S., van den Berg, K., van der Hoeven, G.: Teaching functional programming to first-year students. *Journal of Functional Programming* 3(1), 49–65 (1993)
11. Litecky, C.R., Davis, G.B.: A study of errors, error-proneness, and error diagnosis in Cobol. *Communications of the ACM* 19, 33–38 (1976)
12. Moulton, P.G., Muller, M.E.: Ditrans: a compiler emphasizing diagnostics. *Communications of the ACM* 10, 45–52 (1967)
13. Jones, S.P. (ed.): *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge (2003)
14. van Keeken, P.: Analyzing Helium programs obtained through logging, <http://www.cs.uu.nl/wiki/Hage/MasterStudents>
15. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: POPL 1989: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 60–76. ACM Press, New York (1989)
16. Zelkowitz, M.V.: Automatic program analysis and evaluation. In: Proceedings of the 2nd International Conference on Software Engineering, pp. 158–163. IEEE Computer Society Press, Los Alamitos (1976)

Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude

José Eduardo Rivera¹, Esther Guerra², Juan de Lara³, and Antonio Vallecillo¹

¹ Universidad de Málaga, Spain
{rivera,av}@lcc.uma.es

² Universidad Carlos III de Madrid, Spain
eguerro@inf.uc3m.es

³ Universidad Autónoma de Madrid, Spain
jdelara@uam.es

Abstract. There is a growing need to explicitly represent the behavioral semantics of Modeling Languages in a precise way, something especially important in industrial environments in which simulation and verification are critical issues. Graph transformation provides one way to specify the semantics of Domain Specific Visual Languages (DSVLs), with the advantage of being intuitive and easy to use for the system designer. Even though its theory has been extensively developed during the last 30 years, it has some limitations concerning specific analysis capabilities. On the contrary, Maude is a rewriting logic-based language with very good formal analysis support, but which requires specialized knowledge. In this paper we show how a mapping between graph transformation-based specifications of DSVL semantics and Maude is possible. This allows performing simulation, reachability and model-checking analysis on the models, using the tools and techniques that Maude provides.

1 Introduction

Modeling languages play a cornerstone role in Model-Driven Engineering (MDE) for representing models and metamodels. Modeling languages are normally defined in terms of their abstract and concrete syntax. The abstract syntax is defined by a metamodel, which describes the concepts of the language, the relationships between them, and the structuring rules that constrain the model elements and combinations in order to respect the domain rules. The concrete syntax defines how the metamodel concepts are represented. This metamodeling approach allows the rapid and effective development of languages and their associated tools (e.g., editors).

In the case of Domain-Specific Visual Languages (DSVLs), the concrete syntax is given by assigning graphical representations to the different elements in the metamodel. By using DSVLs, designers are provided with high-level, intuitive notations which allow building models with concepts from the domain, and not from the solution space or target platforms. This kind of modeling languages greatly facilitates modeling in specialized application areas, having the potential to increase quality and productivity.

In addition to describing the syntactic (i.e., structural) aspects of DSVLs, there is the growing need to explicitly represent their behavioral semantics in a precise, intuitive, yet formal way. This is especially important in certain industrial environments in which simulation and verification are critical issues.

One way of specifying behavioral semantics is in terms of visual rules, which prescribe the preconditions of the actions to be triggered and the effects of such actions. Graph Transformation [1,2] is a declarative, rule-based technique that can be effectively used for expressing behavioral semantics. It is gaining increasing popularity due to its graphical form (making rules intuitive) and formal nature (making rules and grammars amenable to formal analysis). It has already been employed to describe the operational semantics of DSVLs [3], taking the advantage that it is possible to use the concrete syntax of the DSVL in the rules, which then become more intuitive to the designer. The most common formalization of graph transformation is using category theory, but current tool support offers limited analysis capabilities: basically detecting rule dependencies and conflicts [1,4]. Moreover, the analysis is further constrained if dealing with attributes.

The need for other kinds of analysis, such as reachability analysis or model checking, has encouraged some authors to define *semantic mappings* [5] between graph transformation and other semantic domains (see, e.g., [3,4,6,7,8,9,10]). This possibility allows one to use the techniques specific to the target semantic domain for analyzing the source models.

In this sense, Maude [11] is a high-level language and a high-performance interpreter and compiler in the OBJ algebraic specification family, which supports rewriting logic specification and programming of systems. The fact that rewriting logic specifications are executable allows to apply a wide range of formal analysis methods and tools [11]. Maude also offers a comprehensive toolkit for automating such specification analysis, efficient enough to be of practical use, and easy to integrate with software development environments such as Eclipse. Maude has already been proposed as a formal notation and environment for specifying and effectively analyzing models and metamodels [12,13,14]. However, one of its drawbacks is that it requires specialized knowledge and expertise, something that may hinder its usability by the average DSVL designer.

To take advantage of both formalisms, in this paper we define a mapping from graph transformation-based specifications of DSVL semantics to the Maude semantic domain. In this approach, graph transformation rules are translated into Maude rewriting rules. This allows to conduct reachability and model-checking analysis using the tools and techniques already available for Maude. Moreover, properties to be fulfilled by reachable models can be visually specified using graph constraints [1,15], which also use the concrete syntax of the DSVL, and translated into Maude to perform the analysis. In addition, the proposal presented here can also serve to provide the Maude specifications of models with a visual notation that can express its rewriting rules in a graphical and more intuitive way.

Paper organization. First, Section 2 introduces graph transformation and graph constraints using a production system example, and Section 3 gives an overview of rewriting logic and Maude. Then, Section 4 presents our encoding of graph transformation rules in Maude and Section 5 shows how to analyse them using the Maude toolkit. Section 6 discusses tool support. Finally, Sections 7 and 8 compare our work with other related proposals and present some conclusions and lines for future work.

2 Graph Transformation and Graph Constraints

In this section we give an intuition on graph transformation and graph constraints by presenting some rules that belong to a simulator of a DSVL for production systems, and some constraints expressing reachability and invariant properties to be analyzed for such systems. Fig. 1 shows the DSVL metamodel. It defines three kinds of machines: assemblers, containers, and generators of different parts. Each generator can produce a certain number of parts, given by its attribute *counter*. Machines can be connected through trays, which in turn can be interconnected and store parts. The number of parts they actually hold is stored in attribute *nelems* up to its maximum *capacity*, which must be greater than zero (see second OCL constraint to the right of the figure). The last two OCL constraints guarantee that the number of elements on a tray is equal to the number of parts that it contains, and never exceeds its capacity. Finally, human operators are needed to transfer parts between interconnected trays, as the first OCL constraint states.

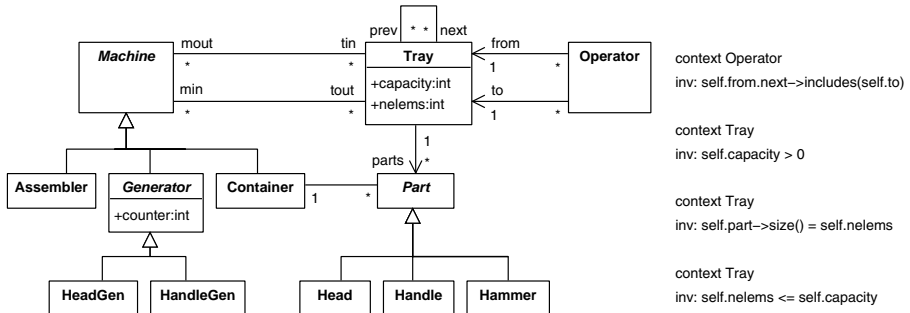


Fig. 1. Metamodel of a DSVL for production systems

Fig. 2 shows a production model example using a visual concrete syntax. It contains four machines (one of each type), four trays and one operator. Machines are represented as decorated boxes, except generators, which are depicted as semi-circles with an icon showing the kind of part they generate. Operators are shown as circles. The model consists of two generators connected to two trays, which are connected to a third one. An operator transfers parts between two of the trays. From the third tray, parts are assembled, deposited in a fourth tray,

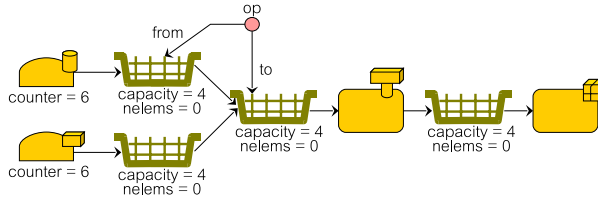


Fig. 2. `initModel` example production system

and finally stored in a container. Although some associations in the metamodel are bidirectional, we have assigned arrows in the concrete syntax, but of course this does not affect navigability.

We use graph transformation systems for the specification of the DSVL behavioral semantics. A graph grammar is made of a set of rules and an initial graph to which the rules are applied. Each rule is made of a left and a right hand side (LHS and RHS) graphs. The LHS expresses pre-conditions for the rule to be applied, whereas the RHS contains the rule's post-conditions. In order to apply a rule to a *host graph*, a morphism (an occurrence or match) of the LHS has to be found in it. If several are found, one is selected randomly. Then, the rule is applied by substituting the match by the RHS. The grammar execution proceeds by applying the rules in non-deterministic order, until none is applicable.

Fig. 3 shows one of the rules describing the DSVL behavioral semantics. The rule specifies the behavior of an assembler machine, which converts one handle and one head into a hammer. The rule is shown in concrete syntax to the left, and in abstract syntax to the right. It can be applied if an occurrence of the LHS is found in the model. Then, the elements in the LHS that do not appear in the RHS are deleted, whereas the elements in the RHS that do not appear in the LHS are created. Our rules may include attribute conditions (which must be satisfied by the match) and attribute computations.

There are two main formalizations of algebraic graph transformation: Double Pushout (DPO) and Single Pushout (SPO). From a practical point of view, their difference is that in DPO, deletion has no side effects. When a node in

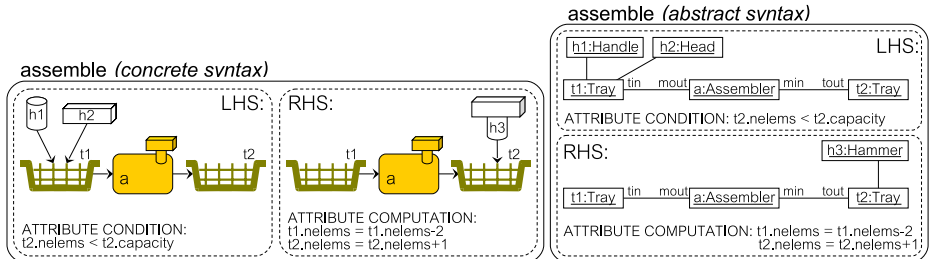


Fig. 3. `Assemble` rule in concrete syntax (left) and abstract syntax (right)

the host graph is deleted by a rule, it can only be connected with those edges explicitly deleted by the rule. For example, if we try applying rule **assemble**, and “h1” is matched to a handle connected to more than one tray (should it be allowed by the metamodel), then the rule cannot be applied at such match as otherwise some edges would become dangling in the host graph. This condition is called *dangling edge* condition. In SPO, dangling edges are removed by the rewriting step. Therefore in DPO, in addition to positive pre-conditions, a LHS also imposes implicit negative pre-conditions in case the rule deletes some node.

A match can be non-injective, which means that two (or more) elements with compatible type in the rule may be matched to a single element in the host graph. If the rule specifies that one of them is deleted and the other one preserved, DPO forbids applying the rule to such a match, while SPO allows its application and deletes both elements. In DPO, this is called the *identification condition*.

Fig. 4 shows further rules for the DSVL simulator. Rule **transferPart** describes the movement of parts through trays, for which the presence of an operator is necessary. This rule uses abstract nodes: part p has an abstract type, what is graphically represented with an asterisk. Of course, no node with an abstract type can be found in the models, but the abstract node in the rule can get instantiated to nodes of any concrete subtype [16]. In this way, rule **transferPart** is equivalent to the three rules resulting from the substitution of part by its children concrete classes.

Rule **moveOperator** models an operator changing to a different pair of connected trays whose source tray has some part waiting to be processed and it is unattended. This last condition is specified by using a negative application condition (NAC) that forbids applying the rule if some operator is already present between the final trays. Note how the match for this rule will be non-injective when applied to the model in Fig. 2 (once the trays have enough pieces), as trays $t2$ and $t4$ in the rule’s LHS will be identified to the same one in the model.

Altogether, graph transformation is an intuitive technique for specifying the behavioral semantics of a DSVL. It also counts with some interesting analysis possibilities. For example, there are results for checking when two rules are conflicting (applying one disables the other) or independent (both can be applied in any order yielding the same result), and to design terminating transformations [1]. However, these analysis techniques are limited as soon as attribute computations or conditions are present in the rules. In this paper we propose

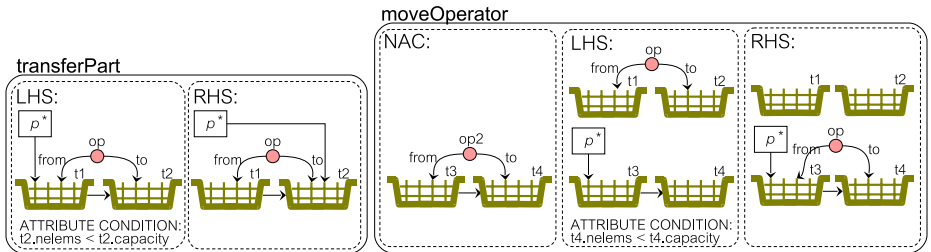


Fig. 4. Additional rules for the DSVL simulator

an automatic mapping into rewriting logic, which allows us to perform different kinds of analysis with Maude. In this way, the designer will have the advantage of a visual and intuitive specification, and an efficient and powerful means for analysis.

From the user perspective, there is a need for an intuitive means of expressing not only the rules but also the model properties to be analyzed. We use graph constraints for such purpose, as they allow employing the concrete syntax of the DSVL.

2.1 Graph Constraints

Graph constraints [1,15] constitute a graphical and formal way to specify model properties. Roughly, a graph constraint is made of a number of graphs related through morphisms, and demands the existence or absence of such graphs in the model. Constraints can be used for several purposes, here we use them in reachability analysis, to select those reachable models satisfying them.

As an example, Fig. 5 shows two graph constraints. The one on the left is made of two graphs, and asks for a positive occurrence of the left graph $\exists \text{ Container}$ for which there is no occurrence of the related right graph $\neg \exists \text{ Parts}$. That is, the constraint checks whether the model has an empty container. For simplicity we restrict to injective morphisms from the constraint to the model. Similar to rules, constraints may contain objects with abstract typing.

The constraint on the right of the figure is made of one graph ($\exists \text{ PartOverflow}$), with an attribute condition specifying that the number of elements in the tray should be bigger than the tray capacity. Thus, the constraint demands the existence of a tray with a number of parts exceeding its capacity. Note how the model in Fig. 2 satisfies the first constraint but not the second. These two constraints will be used later for the specification of properties for reachable states and invariants.

Technically, the constraints we use follow the approach of [15], where atomic constraints are defined by quantification of injective morphisms of the form $a: X \rightarrow C$. This means that a constraint $\exists(a: X \rightarrow C)$ is satisfied by a model G , if for each occurrence of X in G (i.e. $\forall p_i: X \rightarrow G$) there is some occurrence of C related to the occurrence of X ($\exists c_i: C \rightarrow G$ s.t. $p_i = c_i \circ a$). We consider nested constraints [15], i.e. constraints with an extra condition $\exists(a: X \rightarrow C, \exists(b: C \rightarrow C_i))$, which can be either positive ($\exists(b: C \rightarrow C_i)$) or negative ($\neg \exists(b: C \rightarrow C_i)$). Note that the outer-most constraint a has to be positively satisfied, and we emulate the existential quantification of the graph

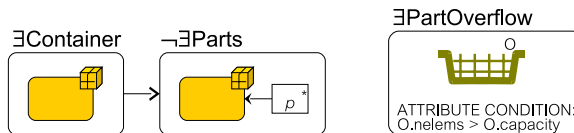


Fig. 5. Examples of graph constraints

C by taking the graph X to be empty. Thus, for example, the left constraint in Fig. 5 is theoretically depicted as $\exists(a: \emptyset \rightarrow \text{Container}, \neg\exists(b: \text{Container} \rightarrow \text{Parts}))$. In this paper, the X graph of the constraint $a: X \rightarrow C$ is always empty, and we visually show the quantifications as if applied to graphs (instead of to morphisms).

3 An Introduction to Maude

Maude [11] is a high-level language and a high-performance system in the OBJ algebraic specification family. It supports membership equational logic and rewriting logic specification and programming of systems. Thus, Maude integrates an equational style of functional programming with rewriting logic computation. Because of its efficient rewriting engine, able to execute more than 3 million rewriting steps per second on standard PCs, and because of its metalanguage capabilities, Maude turns out to be an excellent tool to create executable environments for different logics, models of computation, theorem provers, or even programming languages. Maude has already been successfully used in software engineering tools and several applications [17]. This section introduces those Maude's features necessary for understanding the paper; the interested reader is referred to [11] for further details.

Rewriting logic is a logic of change that can naturally deal with states and nondeterministic concurrent computations. A distributed system is axiomatized by an equational theory describing its set of states and a collection of rewrite rules. Computation in a functional module is accomplished by using the equations as simplification rules from left to right until a canonical form is found.

Maude objects are structures of the form $\langle o : c \mid a_1:v_1, \dots, a_n:v_n \rangle$, where o is the object identifier (of Sort `Obj`), c is the class the object belongs to, a_i are attribute identifiers and v_i their corresponding current values.

While functional modules specify membership equational theories, rewrite theories are specified by *system modules*. A system module may have the same declarations of a functional module plus rules of the form $t \rightarrow t'$, where t and t' are terms. These rules specify the dynamics of a system in rewriting logic. They describe the local, concurrent transitions that are possible in the system, i.e., when a part of the system state fits the pattern t then it can change to a new local state fitting pattern t' . The guards of conditional rules act as blocking pre-conditions, in the sense that a conditional rule can only be fired if the condition is satisfied.

The syntax for conditional rules is `cr1 [l] : t => t' if Cond`, with l the rule label and *Cond* its condition. To illustrate this, the following **BANK** system module defines two classes and a rule. The rule specifies the system behavior when a **Deposit** object that refers to an existing **Account** is found: the **Deposit** object disappears (i.e., it is “consumed” by the rule) and the **balance** of the **Account** object is increased.

```

mod BANK is
  class Account | balance : Int .
  class Deposit | account : Oid, amount : Int .
  vars N M : Nat . vars A D : Oid .
  crl [deposit] :
    < A : Account | balance : N >
    < D : Deposit | account : A, amount : M >
    => < A : Account | balance : N + M >
  if (M > 0) .
endm

```

The `rewrite` command can be used to execute the system, which applies the rules until no further rule can be applied.

```

Maude> rewrite < 'A : Account | balance : 1000 >
          < 'D1 : Deposit | account : 'A, amount : 100 >
          < 'D2 : Deposit | account : 'A, amount : 50 > .
result @Object: < 'A : Account | balance : 1150 >

```

4 From Graph Transformation to Maude

In this section, we show how models and their dynamic behavior (given by graph grammars rules) can be encoded into their corresponding Maude specifications.

4.1 Encoding Models in Maude

Nodes are represented as Maude objects, and node attributes as object attributes. Edges are represented by object attributes too, each one representing the reference (by means of object identifiers) to the target node of the edge. Thus, models are structures of the form $mm\{obj_1\ obj_2\ \dots\ obj_N\}$, where mm is the name of the metamodel, and obj_i are the objects that represent the nodes. For instance, the following Maude specification describes the hammers production model depicted in Fig. 2:

```

ProductionSystem {
  < 'heg : HeadGen | counter : 6, tin : empty, tout : 't1 >
  < 'hag : HandleGen | counter : 6, tin : empty, tout : 't2 >
  < 't1 : Tray | parts : empty, next : 't3, prev : empty, min : 'heg,
    mout : empty, capacity : 4, nelems : 0 >
  < 't2 : Tray | parts : empty, next : 't3, prev : empty, min : 'hag,
    mout : empty, capacity : 4, nelems : 0 >
  < 'op : Operator | from : 't1, to : 't3 >
  < 't3 : Tray | parts : empty, next : empty, prev : ('t2, 't1),
    min : empty, mout : 'a, capacity : 4, nelems : 0 >
  < 'a : Assembler | tin : 't3, tout : 't4 >
  < 't4 : Tray | parts : empty, next : empty, prev : empty, min : 'a,
    mout : 'co, capacity : 4, nelems : 0 >
  < 'co : Container | tin : 't4, tout : empty, parts : empty > }

```


Quoted identifiers are used as object identifiers. Relations with multiplicity ***** are represented as sets of references (of sort **Set{Oid}**), with **empty** as the empty set, and **_,_** the associative and commutative union operator (with identity element **empty**). Relations with arity **0..1** are represented as values of sort **Maybe{Oid}**, which can be either **null** or an object identifier.

Metamodels are encoded using a sort for every metamodel element: sort **@Class** for classes, sort **@Attribute** for attributes, sort **@Reference** for references, etc. Thus, a metamodel is represented by declaring a constant of the corresponding sort for each metamodel element [18]. In addition, other sorts have been defined to represent model elements, such as **@Object** and **@Structural-FeatureInstance**, which represent objects and their features, respectively.

4.2 Encoding Rules in Maude

Dynamic behavior is specified in Maude in terms of rewrite rules added to the corresponding model specification. Thus, graph grammars rules can be naturally translated into Maude rewrite rules. Attribute conditions are encoded as Maude rule conditions, and attribute computations are encoded as Maude computations performed in the right-hand side of the rules. With this, rule **assemble** (shown in Fig. 3) is written in Maude as follows:

```

crl [Assemble] :
  ProductionSystem {
    < H1 : Handle | SFS@H1 > < H2 : Head | SFS@H2 >
    < T1 : Tray | parts : (H2, H1, PARTS@T1), mout : (A, MOUT@T1),
      nelems : NEL@T1, SFS@T1 >
    < A : Assembler | tin : (T1, TIN@A), tout : (T2, TOUT@A), SFS@A >
    < T2 : Tray | min : (A, MIN@T2), capacity : CAP@T2,
      parts : PARTS@T2, nelems : NEL@T2, SFS@T2 > OBJSET }
=> spo(H1 H2,
  ProductionSystem{
    < T1 : Tray | parts : PARTS@T1, mout : (A, MOUT@T1),
      nelems : NEL@T1 - 2, SFS@T1 >
    < A : Assembler | tin : (T1, TIN@A), tout : (T2, TOUT@A), SFS@A >
    < T2 : Tray | min : (A, MIN@T2), capacity : CAP@T2,
      parts : (PARTS@T2, H3), nelems : NEL@T2 + 1, SFS@T2 >
    < H3 : Hammer | > OBJSET })
if H3 := newHammerId() /\ (NEL@T2 < CAP@T2) .

```

Edges that represent bidirectional relationships are encoded as two references: one in each direction. Attributes of LHS nodes used in conditions (such as attribute **capacity** of T2), or in computations (such as attribute **nelems** of T1 and T2), are assigned a variable value to be able to: (a) refer to them in rule conditions; and (b) perform attribute computations in the RHS of Maude rules. Furthermore, new variables (**SFS@H1**, **SFS@H2**, etc.) are included in the object specifications to represent further attributes and edges not described in the LHS or RHS of graph grammar rules. Operation **newHammerId** generates a new object identifier (using a predefined Maude counter [11]) for each created hammer.

The two semantics of graph transformation, DPO and SPO, are realized in Maude by the `dpo` and `spo` operations. Operation `spo` (used above) is placed in the RHS and removes all dangling references from the model when applying the rule. Operation `dpo` (which is placed in the rule condition) checks that there are no more edges connected to the elements that will be deleted than those mentioned in the LHS of the graph grammar rule.

Each negative application condition (NAC) is encoded as an operation placed in the rule condition that checks if an occurrence of the specified NAC pattern (together with the matched LHS) is found in the model. If so, this operation forbids the rule application. To illustrate the encoding of NACs, rule `moveOperator` (Fig. 4) is encoded in Maude as follows:

```

cr1 [MoveOperator] :
  ProductionSystem {
    < T1 : Tray | SFS@T1 >
    < T2 : Tray | SFS@T2 >
    < OP : Operator | from : T1, to : T2, SFS@OP >
    < T3 : Tray | next : (T4, NEXT@T3), parts : (P, PARTS@T3), SFS@T3 >
    < T4 : Tray | prev : (T3, NEXT@T4), capacity : CAP@T4,
      nelems : NEL@T4, SFS@T2 >
    < P : X:Part | SFS@P > OBJSET }
=> ProductionSystem{
  < T1 : Tray | SFS@T1 >
  < T2 : Tray | SFS@T2 >
  < OP : Operator | from : T3, to : T4, SFS@OP >
  < T3 : Tray | next : (T4, NEXT@T3), parts : (P, PARTS@T3), SFS@T3 >
  < T4 : Tray | prev : (T3, NEXT@T4), capacity : CAP@T4,
    nelems : NEL@T4, SFS@T2 >
  < P : X:Part | SFS@P > OBJSET }
if (NEL@T4 < CAP@T4) /\
  LHS := < T1 : Tray | SFS@T1 > < T2 : Tray | SFS@T2 >
  ... < P : X:PartS | SFS@P > /\
  MODEL := ProductionSystem{LHS OBJSET} /\
  not NAC@MoveOperator(LHS, MODEL).

```

Non-injective matches are realized by generating different Maude rules for each match possibility (except those for which the identification condition holds if DPO semantics is used). Thus, rule `moveOperator` generates 15 Maude rules (the fourth Bell number, since we have 4 compatible elements). For instance, the rule in which T2 and T4 represent the same tray is written as follows:

```

cr1 [MoveOperator2] :
  ProductionSystem {
    < T1 : Tray | SFS@T1 >
    < T2 : Tray | capacity : CAP@T2, nelems : NEL@T2,
      prev : (T3, NEXT@T2), SFS@T2 >
    < OP : Operator | from : T1, to : T2, SFS@OP >
    < T3 : Tray | next : (T2, NEXT@T3), parts : (P, PARTS@T3), SFS@T3 >
    < P : X:Part | SFS@P > OBJSET }

```

```

=> ProductionSystem{
  < T1 : Tray | SFS@T1 >
  < T2 : Tray | capacity : CAP@T2, nelems : NEL@T2,
    prev : (T3, NEXT@T2), SFS@T2 >
  < OP : Operator | from : T3, to : T2, SFS@OP >
  < T3 : Tray | next : (T2, NEXT@T3), parts : (P, PARTS@T3), SFS@T3 >
  < P : X:Part | SFS@P > OBJSET }
if (NEL@T2 < CAP@T2) /\ LHS := ... /\ MODEL := ... /\
  not NAC@MoveOperator2(LHS, MODEL) .

```

Finally, graph constraints are translated into Maude in a similar way as the LHS and NACs of the rules. For the moment, we restrict to translating constraints with two levels of nesting at most, e.g. like the one to the left of Fig. 5. We will use these constraints in the next section to specify properties for reachability analysis.

5 Analyzing the Behavioral Semantics with Maude

Once the system specifications are encoded in Maude, we obtain a rewriting logic specification of the system. As these specifications are executable, they can be used to simulate and analyze the system.

In this paper we will focus on three kinds of analysis: *simulation*, to execute the system specifications; *reachability analysis*, to look for deadlocks and to prove system invariants; and *LTL model checking*, to analyze some liveness properties. The analysis will be illustrated with the use of examples. In addition, models can be further analyzed using other available tools in Maude’s formal environment: the LTL satisfiability and tautology checker, the Church-Rosser checker, the coherence checker and the termination tool. We refer the interested reader to [11] for details on these commands and tools.

5.1 Simulation

Maude specifications can be executed using the `rewrite` and `frewrite` commands, which implement two different execution strategies: a top-down rule-fair strategy, and a depth-first position-fair strategy, respectively [11]. Thus, we can execute the model depicted in Fig. 2 by simply typing: “`rewrite initModel .`” The output of this command shows a resulting object configuration where six hammers are created and finally stored:

```

ProductionSystem{
  < 'heg : HeadGen | counter : 0, tin : empty, tout : 't1 >
  < 'hag : HandleGen | counter : 0, tin : empty, tout : 't2 >
  < 't1 : Tray | parts : empty, next : 't3, prev : empty, min : 'heg,
    mout : empty, capacity : 4, nelems : 0 >
  < 't2 : Tray | parts : empty, next : 't3, prev : empty, min : 'hag,
    mout : empty, capacity : 4, nelems : 0 >
  < 'op : Operator | from : 't2, to : 't3 >

```

```

< 't3 : Tray | parts : empty, next : empty, prev : ('t2, 't1),
    min : empty, mout : 'a, capacity : 4, nelems : 0 >
< 'a : Assembler | tin : 't3, tout : 't4 >
< 't4 : Tray | parts : empty, next : empty, prev : empty, min : 'a,
    mout : 'co, capacity : 4, nelems : 0 >
< 'co : Container | tin : 't4, tout : empty,
    parts : ('hammer1, ... 'hammer6) > }
< 'hammer1 : Hammer | empty >
...
< 'hammer6 : Hammer | empty > }

```

The **rewrite** and **frewrite** commands also allow users to specify upper bounds for the number of rule applications. This can be very useful to simulate non-terminating systems, or to perform step-by-step executions.

5.2 Reachability Analysis

Executing the system using the **rewrite** and **frewrite** commands means exploring just one of the possible behaviors of the system. The Maude **search** command allows to explore (following a breadthfirst strategy up to a specified bound) the reachable state space in different ways. The command requires specifying as input the properties that the reachable states have to satisfy, and it returns those states satisfying them. For this purpose we use the graph constraints we introduced in Section 2.1, and then translate them into Maude, according to the procedure described in Section 4.

Reachability analysis can be used for instance to look for deadlock states (i.e., states on which no further rewrite may take place). In our example, we can check for deadlock states in which there is a container with no parts. This latter property is specified using the constraint to the left of Fig. 5, while the fact that we look for terminal states is specified in Maude using the **search** command with the option **=>!**:

```

search [10] initModel =>!
  ProductionSystem { < C0 : Container | parts : empty, SFS > OBJSET } .

```

where variable **C0** is of sort **Obj**, **SFS** is of sort **Set**{**@StructuralFeatureInstance**}, and **OBJSET** is of sort **Set**{**@Object**}. Please note that we have started searching from the **initModel** shown in Fig. 2 and limited the number of solutions to 10 — otherwise we obtain hundreds of solutions. The following is the last of the ten solutions found:

```

Solution 10 (state 8141)
C0 --> 'co
OBJSET -->
< 'heg : HeadGen | counter : 0, tin : empty, tout : 't1 >
< 'hag : HandleGen | counter : 2, tin : empty, tout : 't2 >
< 't1 : Tray | parts : ('head5, 'head3), next : 't3, prev : empty,
    min : 'heg, mout : empty, capacity : 4, nelems : 2 >

```

```

< 't2 : Tray | parts : ('handle4, 'handle3, 'handle2, 'handle1), next : 't3,
  prev : empty, min : 'hag, mout : empty, capacity : 4, nelems : 4) >
< 'op : Operator | from : 't1, to : 't3 >
< 't3 : Tray | parts : ('head6, 'head4, 'head2, 'head1), next : empty,
  prev : ('t2, 't1), min : empty, mout : 'a, capacity : 4, nelems : 4 >
< 'a : Assembler | tin : 't3, tout : 't4 >
< 't4 : Tray | parts : empty, next : empty, prev : empty, min : 'a,
  mout : 'co, capacity : 4, nelems : 0 >
< 'head1 : Head | empty > ... < 'head6 : Head | empty >
< 'handle1 : Head | empty > ... < 'handle4 : Head | empty >
SFS --> tin : 'T4, tout : empty

```

It shows a deadlock situation, caused by the fact that tray 't3 is full of parts of the same type, therefore not allowing the assembler machine to proceed.

In order to display the shortest sequence of rewrites followed to reach that state (state 8141) we can use the command `show path n`, where n is the number of the state. If we are only interested in the labels of the applied rules, the command `show path labels n` can be used instead. In this case, `show path labels 8141` produces the sequence: **GenHead GenHead GenHead GenHead GenHandle GenHandle GenHandle GenHandle TransferPart GenHead TransferPart GenHead TransferPart TransferPart**, which is one of the possible paths leading to the undesirable deadlock state (note that the operator has not moved in this case).

Arrow `=>*` of `search` command implies searching for proofs (that fulfill the search pattern) consisting of none, one or more rewriting steps. To enforce a rewriting proof consisting of exactly one step, arrow `=>1` should be used instead; if rewriting proofs consisting of one or more steps are desired, arrow `=>+` must be used. Note that arrow `=>+` can be especially useful in cases where the initial model fulfills the search pattern, for example when looking for cycles. Finally, arrow `=>!` is used to search only canonical final states, that is, states that cannot be further rewritten. Upper bounds for the number of rule applications and solutions can also be specified in the search. These bounds are very useful in non-terminating systems.

We can also use the `search` command to prove safety properties. For instance, we can check whether starting from `initModel`, the capacity of any tray is exceeded. This property can be visually specified with the constraint shown to the right of Fig. 5, and using the Maude search command `=>*`:

```

search initModel =>*
  ProductionSystem { < 0 : Tray | capacity : C, nelems : N, SFS > OBJSET }
  such that N > C .
No solution.

```

Since no solutions are found, we can state that (starting from `initModel`) the capacities of the trays are never exceeded.

Invariants can be checked by looking for states that do not satisfy them. Whenever an invariant is violated, we obtain a counterexample (up to time and memory limitations). However, when the number of states reachable from the

initial state is infinite, and the invariant holds, the search never terminates. In these cases, bounded search is a widely used procedure that: (a) together with a big depth can greatly increase the confidence that the invariants hold, and (b) can be quite effective in finding counterexamples, where applicable.

5.3 LTL Model Checking

Maude offers a linear temporal logic explicit-state model checker [19], which allows us to check whether every possible behavior starting from a given initial model satisfies a given temporal logic property. It can be used to prove safety and liveness properties of rewriting systems when the set of states reachable from an initial state is finite. Full verification of invariants in infinite-state systems can be accomplished by verifying the invariants on finite-state abstractions [20] of the original infinite-state system, that is, on an appropriate quotient of the original system whose set of reachable states is finite [11].

State predicates are needed to specify safety and liveness properties. For instance, suppose that we want to check if a created hammer is eventually stored in the container. In this case, we need two predicates: **exist** and **stored**, that will determine whether a part exists and is stored, respectively. State predicates are defined in Maude as operators of sort **Prop**, and their semantics is given in terms of equations that specify when the predicates evaluate to true.

For instance, predicates **exist** and **stored**, together with a predicate **operated** that checks whether an operator **O** is located between the trays **T** and **T'**, can be specified as follows:

```
ops exist stored : Oid -> Prop .
op operated : Oid Oid Oid -> Prop .
eq ProductionSystem { < O : C | SFS > OBJSET } |= exist(O) = true .
eq ProductionSystem { < O : Container | parts : (P, PARTS), SFS >
  OBJSET } |= stored(P) = true .
eq ProductionSystem { < O : Operator | from : T, to : T', SFS >
  OBJSET } |= operated(O, T, T') = true .
eq MODEL |= PROP = false [otherwise] .
```

With these predicates we are ready to model-check different LTL properties making use, e.g., of the *eventually* ($\langle \rangle$) and *henceforth* ($[]$) connectives. For instance, we can check whether a given hammer (**'hammer1**) is eventually stored:

```
reduce modelCheck(initModel, [] (exist('hammer1) -> <> stored('hammer1))).
result Bool: true
```

Similarly, the corresponding formula for all possible hammers can be derived [18]. Other interesting properties to model-check are, for instance, that operators eventually move between trays; or that generated parts are eventually consumed to produce hammers:

```
reduce modelCheck(initModel, [] (operated('op, 't1, 't3) ->
  <> operated('op, 't2, 't3)) .
reduce modelCheck(initModel, [] (exist('head1) -> <> ~ exist('head1))).
```

As expected, none of them are fulfilled due to deadlock states, and therefore we get the corresponding counterexamples showing why they fail. With this information, we could very easily modify the dynamic behavior of our model to avoid deadlocks. For example, we could add a condition to the **transferPart** rule so that a part (of type T) is never put in the output tray if all the parts currently in it are of the same type T .

6 Tool Support

We have built a Maude code generator and integrated it in the AToM³ tool [3]. This tool allows the definition of DSVLs by metamodeling and defining model manipulations by graph transformations. Thus, AToM³ is now provided with the analysis capabilities presented in previous section, and, on the other hand, it can be used as a visual front-end to Maude.

The code generator synthesizes Maude specifications from the DSVL meta-model, the graph transformation rules describing its semantics and the models. As in AToM³ the transformation semantics (DPO/SPO) as well as the possibility of non-injective matches can be chosen by the user, this decision has to be made when generating the Maude specification. In the generated environment for the DSVL, the user can visually specify properties by means of graph constraints, which are checked using reachability analysis, starting from the current model.

Fig. 6 shows the generated environment for the example DSVL, where the **initModel** is shown in the background. The lower three buttons to the left allow simulating the model using the graph transformation rules, performing reachability analysis and generating plain Maude code, so that for example model checking can be performed. The figure shows a screenshot during the specification of the reachability analysis. On top of this window, a control dialog is presented

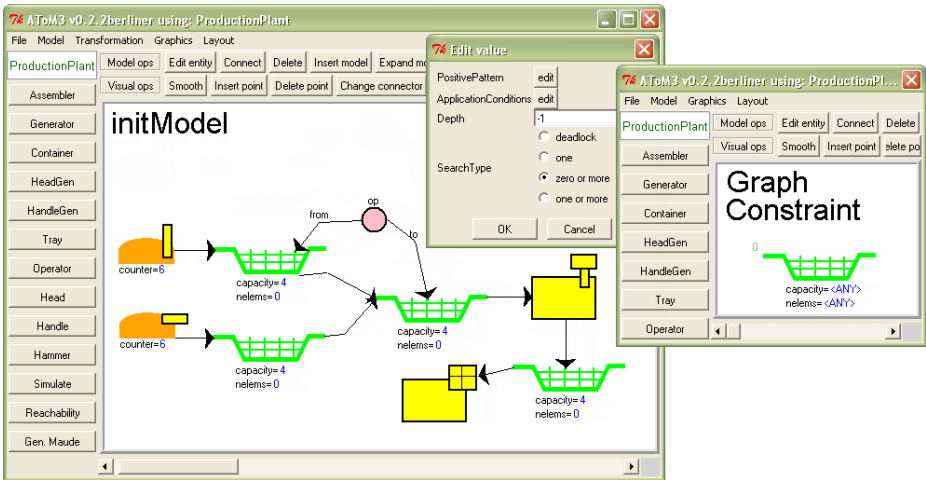


Fig. 6. Reachability analysis in the final environment

that allows to set the graph constraint and the attribute conditions, to select the search type and the number of solutions to be found. The right-most window shows the specification of the graphical constraint presented to the right of Fig. 5. Once the user starts the reachability process, Maude is called, and the possible solution is presented back in the AToM³ tool by using a parser of Maude terms. In this way, the analysis is performed in the background and the user is presented the results in terms of the original DSL, thus hiding the formal methods used for the analysis. Further visual support for defining model checking predicates and showing model execution paths are left for future work.

7 Related Work

One common way to specify the semantics of a language is to define a translation from expressions in that language to expressions in another language with well defined semantics [21]. These *semantic mappings* between semantic domains are very useful not only to provide precise semantics to metamodels, but also to be able to simulate, analyze or reason about them using the logical and semantical framework available in the target domain [5].

The most common formalization of graph transformation is the so-called algebraic approach, which uses category theory to express the rewriting. This approach supports a number of interesting analysis techniques, such as detecting rule dependencies [1] or calculating critical pairs (minimal context of pairs of conflicting rules) [4] — usually with simple type graphs [16].

However, further kinds of analysis require semantic mappings between graph transformation and other semantic domains more appropriate to represent and reason about certain properties, and to conduct certain kinds of analysis, mostly when dealing with attributes. For example, in [7] rules are translated into Alloy in order to study the applicability of sequences of rules and the reachability of models. In [8] rules are transformed into pre- and post-conditions for rule analysis (e.g., conflict detection) using standard OCL tools, and in [6] rules are translated into Petri nets to check safety properties. In a previous work [9] we used Petri nets to analyze reachability, place and transition invariants, boundedness, etc. However these approaches are somehow limited: they do not allow attributes in metamodel elements, or restrict their types to finite domains, and they are only available for systems with very similar semantics to Place-Transition nets.

In [22], the behavioral semantics of diagrammatic languages were specified with collaboration diagrams and formalized as graph transformation rules; however, no explicit verification mechanism was presented.

Regarding model-checking of graph transformation systems, there are few approaches. For instance, in [10], graph grammars are transformed into Abstract State Machines (ASM) and transition systems for model-checking, whereas the Groove tool [23] incorporates a built-in explicit model-checker. Both are restricted in the kind of graphs and attributes they can handle: the former does not allow unbounded attributes in metamodel elements, and the latter works

with simple graphs (i.e. no parallel edges) with edge labels. Interestingly, in [23] a logic for graphs similar to LTL (so called Graph-Based Linear Temporal Logic) is proposed, which extends LTL with constraints similar to the graph constraints presented in section 2.1. Using graph constraints as predicates for LTL is left for future work in our case.

The use of Maude as a semantic domain in which DSLs can be mapped has enabled the use of more powerful analysis on the source models. Thus, with Maude we can deal with more complex semantics description (i.e., not limited to place-transition systems), and conduct reachability and model-checking analysis on the systems, supported by the Maude toolkit. In addition, Maude offers an integrated environment and a toolkit for conducting all the different kinds of analysis, instead of having to define several semantic mappings to different semantic domains — one for each kind of analysis.

Another group of related works includes those that formalize the concepts of concrete metamodeling languages in Maude, namely UML or OCL. For example, RIVIERA [24] is a framework for the verification and simulation of UML class diagrams and statecharts. It is based on the representation of class and state models as terms in Maude modules that specify the UML metamodel; it makes a heavy use of the reflective capabilities of Maude, defining and handling most model operations at the metalevel. MOVA [25] is another Maude-based modeling framework for UML, which provides support for OCL constraint validation, OCL query evaluation, and OCL-based metrication. In MOVA, both UML class and object diagrams are formalized as Maude theories. MOMENT [26,27] is a generic model management framework which uses Maude modules to automatically serialize software artifacts. It supports OCL queries and is also integrated in Eclipse. When compared to our work, these approaches are focused on formalizing UML and OCL, but do not provide explicit means to represent the behavior of models, as we do using Maude rewrite rules.

Finally, other authors have also made some progress formalizing the concepts of concrete metamodeling languages. For example, di Ruscio et al. [28] have formalized KM3 using Abstract State Machines; Boronat and Meseguer [14] propose an algebraic semantics for the MOF metamodeling framework, giving an explicit formal representation for each of the different notions that may be involved in a metamodeling framework: model type, metamodel realization, and metamodel conformance; finally, Poernomo [29] uses Constructive Type Theory and a related synthesis formal method known as proofs-as-programs to formalize MOF-based metamodels and model transformations, and to develop model transformations that are provably correct with respect to a given functional specification. These works are complementary to ours, since they aim at formalizing the concepts of metamodeling frameworks and notations, without explicit means to specify behavior, whilst our goal is to allow the explicit specification of the behavioral aspects of models using a visual language, and to provide the mappings to the corresponding Maude (behavioral) specifications in order to make them amenable to formal analysis and reasoning using the Maude toolkit.

8 Conclusions and Future Work

In this paper we have presented an encoding in Maude of the graph transformation rules that define the behavioral semantics for DSVLs. This semantic mapping has enabled the formal analysis of the source models using the capabilities of Maude for executing the specifications, and for conducting reachability and model-checking analysis.

The use of Maude as a semantic domain in which to map DSVLs has opened the path to further kinds of analysis of rule-based DSVLs, such as the study of termination and confluence of the specifications [30], or the use of *strategies* [11] for determining the order in which the graph transformation rules are selected and executed. These two are interesting lines of work that we can plan to address shortly, and whose findings can be applicable to other rule-based notations for modeling systems. Scalability issues, as well as a detailed comparison of efficiency with respect to approaches like [10] and [23] are also left for future work.

We are also currently working on further extending the back-annotation of results of the analysis in Maude, in order to visualize them using the DSVL native notation, i.e., making the use of Maude completely transparent to the user. Although this is already done for reachability analysis, we are also looking at the specification of the predicates to be model-checked, using the DSVL graphical editor, as well as to show execution paths through states.

Acknowledgements. The authors would like to thank the anonymous referees for their insightful comments and suggestions. This work has been partially supported by Spanish Research Projects TSI2005-08225-C07-06, TIN2006-09678, TIN2005-09405-C02-01, P07-TIC-03184 and TIN2008-00889-E.

References

1. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation. Springer, Heidelberg (2006)
2. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations. Foundations, vol. 1. World Scientific, Singapore (1997)
3. de Lara, J., Vangheluwe, H.: Defining visual notations and their manipulation through meta-modelling and graph transformation. *Journal of Visual Languages and Computing* 15(3–4), 309–330 (2006)
4. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 161–176. Springer, Heidelberg (2002)
5. Vallecillo, A.: A journey through the secret life of models. In: Model Engineering of Complex Systems (MECS). Number 08331 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany (2008), <http://drops.dagstuhl.de/opus/volltexte/2008/1601>
6. Baldan, P., Corradini, A., König, B.: A static analysis technique for graph transformation systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 381–395. Springer, Heidelberg (2001)

7. Baresi, L., Spoletini, P.: On the use of alloy to analyze graph transformation systems. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 306–320. Springer, Heidelberg (2006)
8. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Analysing graph transformation rules through OCL. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 229–244. Springer, Heidelberg (2008)
9. de Lara, J., Vangheluwe, H.: Translating model simulators to analysis models. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 77–92. Springer, Heidelberg (2008)
10. Varró, D.: Automated formal verification of visual modeling languages by model checking. *Software and System Modeling* 3(2), 85–113 (2004)
11. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
12. Rivera, J.E., Vallecillo, A.: Adding behavioral semantics to models. In: Proc. of EDOC 2007, pp. 169–180. IEEE Computer Society, Los Alamitos (2007)
13. Romero, J.R., Rivera, J.E., Durán, F., Vallecillo, A.: Formal and tool support for model driven engineering with Maude. *Journal of Object Technology* 6(9), 187–207 (2007)
14. Boronat, A., Meseguer, J.: An algebraic semantics for MOF. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 377–391. Springer, Heidelberg (2008)
15. Habel, A., Pennemann, K.: Satisfiability of high-level conditions. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 430–444. Springer, Heidelberg (2006)
16. de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed graph transformation with node type inheritance. *Theoretical Computer Science* 376(3), 139–163 (2007)
17. Martí-Oliet, N., Meseguer, J.: Rewriting logic: roadmap and bibliography. *Theoretical Computer Science* 285(2), 121–154 (2002)
18. Rivera, J.E., Vallecillo, A., Durán, F.: Formal specification and analysis of Domain Specific Languages using Maude. Technical report, Universidad de Málaga (2008), <http://atenea.lcc.uma.es/images/e/eb/AnalysingModels.pdf>
19. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: Proc. WRLA 2002, Pisa, Italy. ENTCS, vol. 71, pp. 115–142. Elsevier, Amsterdam (2002)
20. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. In: Baader, F. (ed.) CADE 2003. LNCS, vol. 2741, pp. 2–16. Springer, Heidelberg (2003)
21. Harel, D., Rumpe, B.: Meaningful modeling: What’s the semantics of “semantics”? *Computer* 37(10), 64–72 (2004)
22. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 323–337. Springer, Heidelberg (2000)
23. Rensink, A.: Explicit state model checking for graph grammars. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 114–132. Springer, Heidelberg (2008)
24. Sáez, J., Toval, A., Fernández Alemán, J.L.: Tool support for transforming UML models to a formal language. In: Proc. of WTUML 2001, pp. 111–115 (2001)
25. The MOVA Group: The MOVA tool: a validation tool for UML (2006), <http://maude.sip.ucm.es/mova/>

26. Boronat, A., Carsí, J.Á., Ramos, I.: Automatic support for traceability in a generic model management framework. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 316–330. Springer, Heidelberg (2005)
27. The ISSI Research Group: MOMENT (2008), <http://moment.dsic.upv.es>
28. Ruscio, D.D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for supporting dynamic semantics specifications of DSLs. Technical Report 06.02, Laboratoire d'Informatique de Nantes-Atlantique (LINA), Nantes, France (2006)
29. Poernomo, I.: Proofs-as-model-transformations. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 214–228. Springer, Heidelberg (2008)
30. Clavel, M., Durán, F., Hendrix, J., Lucas, S., Meseguer, J., Ölveczky, P.C.: The maude formal tool environment. In: Mossakowski, T., Montanari, U., Haverlaen, M. (eds.) CALCO 2007. LNCS, vol. 4624, pp. 173–178. Springer, Heidelberg (2007)

Parse Table Composition

Separate Compilation and Binary Extensibility of Grammars

Martin Bravenboer¹ and Eelco Visser²

¹ University of Oregon, USA
martin.bravenboer@acm.org

² Delft University of Technology, The Netherlands
visser@acm.org

Abstract. Module systems, separate compilation, deployment of binary components, and dynamic linking have enjoyed wide acceptance in programming languages and systems. In contrast, the syntax of languages is usually defined in a non-modular way, cannot be compiled separately, cannot easily be combined with the syntax of other languages, and cannot be deployed as a component for later composition. Grammar formalisms that do support modules use whole program compilation.

Current extensible compilers focus on source-level extensibility, which requires users to compile the compiler with a specific configuration of extensions. A compound parser needs to be generated for every combination of extensions. The generation of parse tables is expensive, which is a particular problem when the composition configuration is not fixed to enable users to choose language extensions.

In this paper we introduce an algorithm for *parse table composition* to support separate compilation of grammars to *parse table components*. Parse table components can be composed (linked) efficiently at runtime, i.e. just before parsing. While the worst-case time complexity of parse table composition is exponential (like the complexity of parse table generation itself), for realistic language combination scenarios involving grammars for real languages, our parse table composition algorithm is an order of magnitude faster than computation of the parse table for the combined grammars.

1 Introduction

Module systems, separate compilation, deployment of binary components, and dynamic linking have enjoyed wide acceptance in programming languages and systems. In contrast, the syntax of languages is usually defined in a non-modular way, cannot be compiled separately, cannot easily be combined with the syntax of other languages, and cannot be deployed as a component for later composition. Grammar formalisms that do support modules use whole program compilation and deploy a compound parser. In this paper we introduce an algorithm for *parse table composition* to support separate compilation of grammars to *parse table components*.

The lack of methods for deploying the definition and implementation of languages as components is harming programming practices. Languages are combined in an undisciplined and uncontrolled way, for example by using SQL, HQL, Shell commands, XPath,

```
import table person [ id INT, name VARCHAR, age INT ];
connection c = "jdbc:postgresql:mybook";
ResultSet rs = using c query { SELECT name FROM person WHERE age > {limit}};
```

Fig. 1. SQL extension of Java in ableJ (Silver)

```
$name = $_GET['name'];
$q = "SELECT * FROM users WHERE name = '" . $name . "'";
$q = <| SELECT * FROM users WHERE name = ${$name} |>;
system("svn cat \"file name\" -r" . $rev);
system(<| svn cat "file name" -r${$rev} |>);
```

Fig. 2. SQL and Shell extensions of PHP (StringBorg)

```
class FileEditor {
  void handle(Event e) when e@Open { ... }
  void handle(Event e) when e@Save { ... }
  void handle(Event e) {...} }
```

Fig. 3. Predicate dispatch in JPred (Polyglot)

```
Return(ConditionalExpression(e1, e2, e3)) -> If(e1, Return(e2), Return(e3))
[[ return e1 ? e2 : e3; ]] -> [[ if($e1) return $e2; else return $e3; ]]
```

Fig. 4. Transformation with concrete Java syntax (Stratego)

regular expressions, and LDAP in string literals. The compiler of the host language has no knowledge at all of these languages. Hence, the compiler cannot check if the programs are syntactically correct, nor can the compiler help protect the program against security vulnerabilities caused by user input that is not properly escaped. Extensible compilers such as ableJ [1] (based on Silver [2]), JastAddJ [3], and Polyglot [4] support the modular extension of the base language with new language features or embeddings of domain-specific languages. For example, the security vulnerabilities caused by the use of string literals can be avoided by extending the compiler to understand the syntax of the embedded languages. The extended compiler compiles the embedded fragments with the guarantee that user input is properly escaped according to the escaping rules of the embedded language. Figure 1 shows an application of the extensible compiler ableJ. This extension introduces constructs for defining database schemas and executing SQL queries. The implementation of the extension is modular, i.e. the source code of the base Java compiler is not modified. Silver and the ableJ compiler have also been applied to implement extensions for complex numbers, algebraic datatypes, and computational geometry. Similar to the SQL extension of ableJ, the StringBorg syntactic preprocessor [5] supports the embedding of languages in arbitrary base languages to prevent security vulnerabilities. Figure 2 shows applications of embeddings of SQL and Shell commands in PHP. In both cases, the StringBorg compiler guarantees that embedded sentences are syntactically correct and that strings are properly escaped, as opposed to the unhygienic string concatenation on the previous lines. The

implementations of these extensions are modular, e.g. the grammar for the embedding of Shell in PHP is a module that imports grammars of PHP and Shell. Finally, the Polyglot [4] compiler has been used for the implementation of many language extensions, for example Jedd's database relations and binary decision diagrams [6] and JPred's predicate dispatch [7]. Figure 3 illustrates the JPred extension.

Similar to the syntax extensions implemented using extensible compilers, several metaprogramming systems feature an extensible syntax. Metaprograms usually manipulate programs in a structured representation, but writing code generators and transformations in the abstract syntax of a language can be very unwieldy. Therefore, several metaprogramming systems [8,9,10,11] support the embedding of an object language syntax in the metalanguage. The embedded code fragments are parsed statically and translated to a structured representation, thus providing a concise, familiar notation to the metaprogrammer. Figure 4 illustrates this with a transformation rule for Java, which lifts conditional expressions from return statements. The first rule is defined using the abstract syntax of Java, the second uses the concrete syntax. Metaprogramming systems often only require a grammar for the object language, i.e. the compilation of the embedded syntax is generically defined [8,10].

Extensibility and Composition. Current extensible compilers focus on *source-level extensibility*, which requires users to compile the compiler with a specific configuration of extensions. Thus, every extension or combination of extensions results in a different compiler. Some recent extensible compilers support composition of extensions by specifying language extensions as attribute grammars [1,12] or using new language features for modular, type-safe scalable software composition [13,14]. In contrast to the extensive research on composition of later compiler phases, the grammar formalisms used by current extensible compilers do not have advanced features for the modular definition of syntax. They do not support separate compilation of grammars and do not feature a method for deploying grammars as components. Indeed, for the parsing phase of the compiler, a compound parser needs to be generated for every combination of extensions using *whole program compilation*.

Similarly, in metaprogramming systems with support for concrete syntax, grammars for a particular combination of object language embeddings are compiled together with the grammar of the metalanguage into a single parse table. Metaprograms often manipulate multiple object languages, which means that parse tables have to be deployed for all these combinations. The implementation of the later compiler phases is often generic in the object language, therefore the monolithic deployment of parse tables is the remaining obstacle to allowing the *user* to select language extensions.

A further complication is that the base language cannot evolve independently of the extensions. The syntax of the base language is deployed as part of the language extension, so the deployed language extension is bound to a specific version of the base language. Language extensions should only depend on an *interface* of the base language, not a particular implementation or revision.

As a result, extensions implemented using current extensible compilers cannot be deployed as a plugin to the extensible compiler, thus not allowing the *user* of the compiler to select a series of extensions. For example, it is not possible for a user to select a series of Java extensions for ableJ (e.g. SQL and algebraic datatypes) or Polyglot (e.g. JMatch

and Jedd) without compiling the compiler. Third parties should be able to deploy language extensions that do not require the compiler (or programming environment) to be rebuilt. Therefore, methods for deploying languages as *binary components* are necessary to leverage the promise of extensible compilers. We call this *binary extensibility*. One of the challenges in realizing binary extensible compilers is binary extensibility of the syntax of the base language. Most extensible compilers use an LR parser, therefore this requires the introduction of LR parse table components.

Parse Table Composition. In this paper we introduce an algorithm for *parse table composition* to support separate compilation of grammars to *parse table components*. Parse table components can be composed (linked) efficiently at runtime (i.e. just before parsing) by a minimal reconstruction of the parse table. This algorithm can be the foundation for separate compilation in parser generators, load-time composition of parse table components (cf. dynamic linking), and even runtime extension of parse tables for self-extensible parsers [15]. As illustrated by our AspectJ evaluation, separate compilation of modules can even improve the performance of a whole program parser generator. Using parse table composition extensible compilers can support binary extensibility (at least for syntax) by deploying the parser of the base language as a parse table component. The syntax of language extensions can be deployed as binary components plugging into the extensible compiler by providing a parse table component generated for the language extension only.

While the worst-case time complexity of parse table composition is exponential (like the complexity of parse table generation itself), for realistic language combination scenarios involving grammars for real languages, our parse table composition algorithm is an order of magnitude faster than computation of the parse table for the combined grammars, making online language composition one step closer to reality. The goal is to make composition fast enough to make the user unaware of the parse table composition. This will allow composition of the components at every invocation of a compiler, similar to dynamic linking of executable programs and libraries.

We have implemented parse table composition in a prototype that generates parse tables for scannerless [16] generalized LR (GLR) [17,18] parsers. It takes SDF [19] grammars as input. The technical contributions of this work are:

- The idea of parse table composition as symmetric composition of parse tables as opposed to incrementally adding productions, as done in work on incremental parser generation [20,21,22]
- A formal foundation for parse table modification based on automata
- An efficient algorithm for partial reapplication of NFA to DFA conversion, the key idea of parse table composition

2 Grammars and Parsing

A context-free grammar G is a tuple $\langle \Sigma, N, P \rangle$, with Σ a set of terminal symbols, N a set of nonterminal symbols, and P a set of productions of the form $A \rightarrow \alpha$, where we use the following notation: V for the set of symbols $N \cup \Sigma$; A, B, C for variables ranging over N ; X, Y, Z for variables ranging over V ; a, b for variables ranging over Σ ; v, w, x for

variables ranging over Σ^* ; and α, β, γ for variables ranging over V^* . The context-free grammar $1 = \langle \Sigma, N, P \rangle$ will be used throughout this paper, where

$$\Sigma = \{+, N\} \quad N = \{E, T\} \quad P = \{E \rightarrow E + T, E \rightarrow T, T \rightarrow N\} \quad (1)$$

The relation \Rightarrow on V^* defines the derivation of strings by applying productions, thus defining the language of a grammar in a generative way. For a grammar G we say that $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma \in P(G)$. A series of zero or more derivation steps from α to β is denoted by $\alpha \Rightarrow^* \beta$. The relation \Rightarrow_{rm} on V^* defines rightmost derivations, i.e. where only the rightmost nonterminal is replaced. We say that $\alpha A w \Rightarrow_{rm} \alpha \gamma w$ if $A \rightarrow \gamma \in P(G)$. If $A \Rightarrow_{rm}^* \alpha$ then we say that α is a right-sentential form for A .

LR Parsing. An LR(0) parse table is a tuple $\langle Q, \Sigma, N, \text{start}, \text{action}, \text{goto}, \text{accept} \rangle$ with Q a set of states, Σ a set of terminal symbols, N a set of nonterminal symbols, $\text{start} \in Q$, action a function $Q \times \Sigma \rightarrow \text{action}$ where action is either shift q or reduce $A \rightarrow \alpha$, goto a function $Q \times N \rightarrow Q$, and finally $\text{accept} \subseteq Q$, where we use the following additional notation: q for variables ranging over Q ; and S for variables ranging over $\mathcal{P}(Q)$.

An LR parser [23,24] is a transition system with as configuration a stack of states and symbols $q_0 X_1 q_1 X_2 q_2 \dots X_n q_n$ and an input string v of terminals. The next configuration of the parser is determined by reading the next terminal a from the input v and peeking at the state q_n at the top of the stack. The entry $\text{action}(q_n, a)$ indicates how to change the configuration. The entries of the action table are shift or reduce actions, which introduce state transitions that

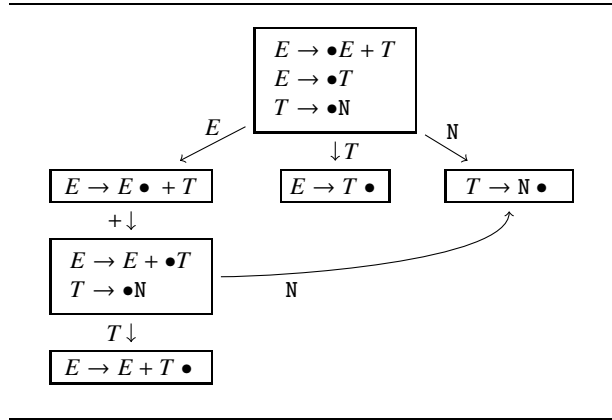


Fig. 5. LR(0) DFA for grammar 1

are recorded on the stack. A shift action removes the terminal a from the input, which corresponds to a step of one symbol in the right-hand sides of a set of productions that is currently expected. A reduce action of a production $A \rightarrow X_1 \dots X_k$ removes $2k$ elements from the stack resulting in state q_{n-k} being on top of stack. Next, the reduce action pushes A and the new current state on the stack, which is determined by the entry $\text{goto}(q_{n-k}, A)$.

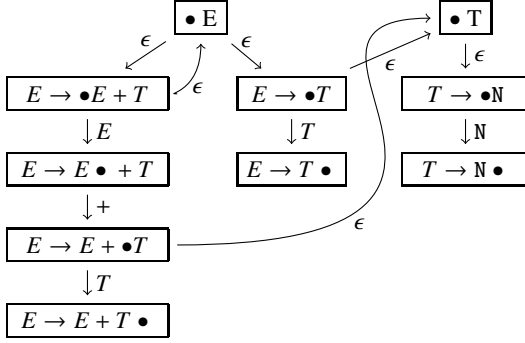
The symbols on the stack of an LR parser are always a prefix of a right-sentential form of a grammar. The set of possible prefixes on the stack is called the *viable prefixes*. We do not discuss the LR parsing algorithm in further detail, since we are only interested in the generation of the action and goto tables.

<pre> function generate-tbl(A, G) = 1 $\langle Q, \delta, \text{start} \rangle := \text{generate-dfa}(A, G)$ 2 for each $q \rightarrow_X q' \in \delta$ 3 if $X \in \Sigma(G)$ then $\text{action}(q, X) := \text{shift } q'$ 4 if $X \in N(G)$ then $\text{goto}(q, X) := q'$ 5 for each $q \in Q$ 6 if $[A \rightarrow \alpha \bullet \text{eof}] \in q$ then 7 $\text{accept} := \text{accept} \cup \{q\}$ 8 for each $[A \rightarrow \alpha \bullet] \in q$ 9 for each $a \in \Sigma(G)$ 10 $\text{action}(q, a) := \text{reduce } A \rightarrow \alpha$ 11 return $\langle Q, \Sigma(G), N(G), \text{start}, \text{action}, \text{goto}, \text{accept} \rangle$ function move(q, X) = 1 return $\{ [A \rightarrow \alpha X \bullet \beta] \mid [A \rightarrow \alpha \bullet X \beta] \in q \}$ </pre>	<pre> function generate-dfa(A, G) = 1 $\text{start} := \text{closure}(\{[S' \rightarrow \bullet A \text{eof}]\}, G)$ 2 $Q := \{\text{start}\}, \delta := \emptyset$ 3 repeat until Q and δ do not change 4 for each $q \in Q$ 5 for each $X \in \{Y \mid [B \rightarrow \alpha \bullet Y \beta] \in q\}$ 6 $q' := \text{closure}(\text{move}(q, X), G)$ 7 $Q := Q \cup \{q'\}$ 8 $\delta := \delta \cup \{q \rightarrow_X q'\}$ 9 return $\langle Q, \delta, \text{start} \rangle$ function closure(q, G) = 1 repeat until q does not change 2 for each $[A \rightarrow \alpha \bullet B \beta] \in q$ 3 $q := q \cup \{ [B \rightarrow \bullet \gamma] \mid B \rightarrow \gamma \in P(G) \}$ 4 return q </pre>
---	--

Fig. 6. LR(0) parse table generation for grammar G

Generating LR Parse Tables. The action and goto table of an LR parser are based on a deterministic finite automaton (DFA) that recognizes the viable prefixes for a grammar. The DFA for recognizing the viable prefixes for grammar 1 is shown in Figure 5. Every state of the DFA is associated to a set of items, where an item $[A \rightarrow \alpha \bullet \beta]$ is a production with a dot (\bullet) at some position of the right-hand side. An item indicates the progress in possibly reaching a configuration where the top of the stack consists of $\alpha\beta$. If the parser is in a state q where $[A \rightarrow \alpha \bullet \beta] \in q$ then the α portion of the item is currently on top of the stack, implying that a string derivable from α has been recognized and a string derivable from β is predicted. For example, the item $[E \rightarrow E + \bullet T]$ represents that the parser has just recognized a string derivable from $E +$. We say that an item $[A \rightarrow \alpha \bullet X \beta]$ *predicts* the symbol X .

To deal with increasingly larger classes of grammars, various types of LR parse tables exist, e.g. LR(0), SLR, LALR, and LR(1). The LR(0), SLR, and LALR parse tables all have the same underlying DFA, but use increasingly more precise conditions on the application of reduce actions. Figure 6 shows the algorithm for generating LR(0) parse tables. The main function `generate-tbl` first calls `generate-dfa` to construct a DFA. The function `generate-dfa` collects states as sets of items in Q and edges between the states in δ . The start state is based on an initial item for the start production. For each set of items, the function `generate-dfa` determines the outgoing edges by applying the function `move` to all the predicted symbols of an item set. The function `move` computes the *kernel* of items for the next state q' based on the items of the current state q by shifting the \bullet over the predicted symbol X . Every kernel is extended to a closure using the function `closure`, which adds the initial items of all the predicted symbols to the item set. Because LR(0), SLR, and LALR parse tables have the same DFA the function `generate-dfa` is the same for all these parse tables. The LR(0) specific function `generate-tbl` initializes the action and goto tables based on the set of item sets. Edges labelled with a terminal become shift actions. Edges labelled with a nonterminal are entries of the goto table. Finally, if there is a final item, then for all terminal symbols the action is a reduce of this production.

Fig. 7. LR(0) ϵ -NFA for grammar 1

```

function generate-nfa( $G$ ) =
1   $Q := \{\bullet A \mid A \in N(G)\}$ 
2  for each  $A \rightarrow \alpha \in P(G)$ 
3     $q := \{[A \rightarrow \bullet \alpha]\}$ 
4     $Q := Q \cup \{q\}$ 
5     $\delta := \delta \cup \{A \rightarrow \bullet \alpha\}$ 
6  repeat until  $Q$  and  $\delta$  do not change
7    for each  $q = \{[A \rightarrow \alpha \bullet X \beta]\} \in Q$ 
8       $q' := \{[A \rightarrow \alpha X \bullet \beta]\}$ 
9       $Q := Q \cup \{q'\}$ 
10      $\delta := \delta \cup \{q \rightarrow_X q'\}$ 
11   for each  $q = \{[A \rightarrow \alpha \bullet B \gamma]\} \in Q$ 
12      $\delta := \delta \cup \{q \rightarrow_\epsilon \bullet B\}$ 
13  return  $\langle Q, \Sigma(G) \cup N(G), \delta \rangle$ 

```

Fig. 8. LR(0) ϵ -NFA generation

Parse Table Conflicts. LR(0) parsers require every state to have a deterministic action for every next terminal in the input stream. There are not many languages that can be parsed using an LR(0) parser, yet we focus on LR(0) parse tables for now. The first reason is that the most important solution for avoiding conflicts is restricting the application of reduce actions, e.g. using the SLR algorithm. These methods are orthogonal to the generation and composition of the LR(0) DFA. The second reason is that we target a generalized LR parser [17,18], which already supports arbitrary context-free grammars by allowing a *set* of actions for every terminal. The alternative actions are performed pseudo-parallel, and the continuation of the parsing process will determine which action was correct. Our parse table composition method can also be applied to target deterministic parsers, but the composition of deterministic parse tables might result in new conflicts, which have to be reported or resolved.

3 LR Parser Generation: A Different Perspective

The LR(0) parse table generation algorithm is very non-modular due to the use of the closure function, which requires all productions of a nonterminal to be known at parse table generation time. As an introduction to the solution for separate compilation of grammars, we discuss a variation of the LR(0) algorithm that first constructs a non-deterministic finite automaton (NFA) with ϵ -transitions (ϵ -NFA) and converts the ϵ -NFA into an LR(0) DFA in a separate step using the subset construction algorithm [25,26]. The ingredients of this algorithm and the correspondence to the one discussed previously naturally lead to the solution to the modularity problem of LR(0) parse table generation.

Generating LR(0) ϵ -NFA. An ϵ -NFA [27] is an NFA that allows transitions on ϵ , the empty string. Using ϵ -transitions an ϵ -NFA can make a transition without reading an input symbol. An ϵ -NFA A is a tuple $\langle Q, \Sigma, \delta \rangle$ with Q a set of states, Σ a set of symbols, and δ a transition function $Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$, where we use the following notation: q for variables ranging over Q ; S for variables ranging over $\mathcal{P}(Q)$, but ranging over Q_D for a DFA D ; X for variables ranging over Σ ; and $q_0 \rightarrow_X q_1$ for $q_1 \in \delta(q_0, X)$.

Figure 7 shows the LR(0) ϵ -NFA for the example grammar 1 (observe the similarity to a syntax diagram). For every nonterminal A of the grammar, there is a *station state* denoted by $\bullet A$. All other states contain just a single LR item. The station states have ϵ -transitions to all the initial items of their productions. If an item predicts a nonterminal A , then there are two transitions: an ϵ -transition to the station state of A and a transition to the item resulting from shifting the dot over A . For an item that predicts a terminal, there is just a single transition to the next item.

Figure 8 shows the algorithm for generating the LR(0) ϵ -NFA for a grammar G . Note that states are singleton sets of an item or just a dot before a nonterminal (the station states). The ϵ -NFA of a grammar G accepts the same language as the DFA generated by the algorithm of Figure 6, i.e. the language of viable prefixes.

Eliminating ϵ -Transitions. The ϵ -NFA can be turned into a DFA by eliminating the ϵ -transitions using the subset construction algorithm [27,24], well-known from automata theory and lexical analysis. Figure 9 shows the algorithm for converting an ϵ -NFA to a DFA. The function ϵ -closure extends a given set of states S to include all the states reachable through ϵ -transitions. The function move determines the states reachable from a set of states S through transitions on the argument X . The function labels is a utility function that returns the symbols (which does not include ϵ) for which there are transitions from the states of S . The main function ϵ -subset-construction drives the construction of the DFA. For every state $S \subseteq Q_E$ it determines the new subsets of states reachable by transitions from states in S .

Applying ϵ -subset-construction to the ϵ -NFA of Figure 7 results in the DFA of Figure 5. This is far from accidental because the algorithm for LR(0) DFA generation of Figure 6 has all the elements of the generation of an ϵ -NFA followed by subset construction. The ϵ -closure function corresponds to the function closure , because ϵ -NFA states whose item predicts a nonterminal have ϵ -transitions to the productions of this nonterminal via the station state of the nonterminal. The first move function constructs the kernel of the next state by moving the dot, whereas the new move function constructs the kernel by following the transitions of the NFA. Incidentally, these transitions exactly correspond to moving the dot, see line 8 of Figure 8. Finally, the main driver function generate-dfa is basically equivalent to the function ϵ -subset-construction. Note that most textbooks call the closure function from the move function, but to emphasize the similarity we moved this call to the callee of move .

```

function  $\epsilon$ -subset-construction( $A, \langle Q_E, \Sigma, \delta_E \rangle$ ) =
1   $Q_D := \{\epsilon\text{-closure}(\{\bullet A\}, \delta_E)\}$ 
2   $\delta_D := \emptyset$ 
3  repeat until  $Q_D$  and  $\delta_D$  do not change
4    for each  $S \in Q_D$ 
5      for each  $X \in \text{labels}(S, \delta_E)$ 
6         $S' := \epsilon\text{-closure}(\text{move}(S, X, \delta_E), \delta_E)$ 
7         $Q_D := Q_D \cup \{S'\}$ 
8         $\delta_D := \delta_D \cup \{S \rightarrow_X S'\}$ 
9  return  $\langle Q_D, \Sigma, \delta_D \rangle$ 

function  $\epsilon$ -closure( $S, \delta$ ) =
1  repeat until  $S$  does not change
2     $S := S \cup \{q_1 \mid q_0 \in S, q_0 \rightarrow_\epsilon q_1 \in \delta\}$ 
3  return  $S$ 

function  $\text{labels}(S, \delta) =$ 
1  return  $\{X \mid q_0 \in S, q_0 \rightarrow_X q_1 \in \delta\}$ 

function  $\text{move}(S, X, \delta) =$ 
1  return  $\{q_1 \mid q_0 \in S, q_0 \rightarrow_X q_1 \in \delta\}$ 

```

Fig. 9. Subset construction algorithm from ϵ -NFA E to DFA D

4 Composition of LR(0) Parse Tables

We discussed the ϵ -NFA variation of the LR(0) parse table generation algorithm to introduce the ingredients of parse table composition. LR(0) ϵ -NFA's are much easier to compose than LR(0) DFA's. A naive solution to composing parse tables would be to only construct ϵ -NFA's for every grammar at parse table generation-time and at composition-time merge all the station states of the ϵ -NFA's and run the subset construction algorithm. Unfortunately, this will not be very efficient because subset construction is the expensive part of the LR(0) parse table generation algorithm. The ϵ -NFA's are in fact not much more than a different representation of a grammar, comparable to a syntax diagram.

The key to an efficient solution is to apply the DFA conversion to the individual parse table components at generation-time, but also preserve the ϵ -transitions as meta-data in the resulting automaton, which we refer to as an ϵ -DFA. The ϵ -transitions of the ϵ -DFA can be ignored as long as the automaton is not modified, hence the name ϵ -DFA, though there is no such thing as a deterministic automaton with ϵ -transitions in automata theory. The ϵ -transitions provide the information necessary for reconstructing a correct DFA using subset construction if states (corresponding to new productions) are added to the ϵ -DFA. In this way the DFA is computed per component, but the subset construction can be rerun partially where necessary. The amount of subset reconstruction can be reduced by making use of information on the nonterminals that overlap between parse table components. Also, due to the subset construction applied to each component, many states are already part of the set of ϵ -NFA states that corresponds to a DFA state. These states do not have to be added to a subset again.

Figure 10a shows the ϵ -DFA for grammar 1, generated from the ϵ -NFA of Figure 7. The E and T arrows indicate the closures of the station states for these nonterminals. The two dashed ϵ -transitions

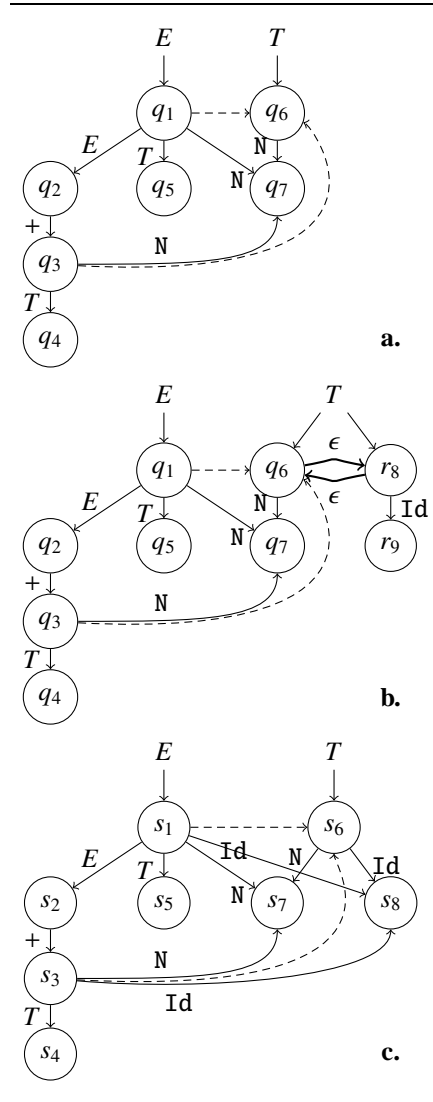


Fig. 10. a. LR(0) ϵ -DFA for grammar 1 b. Combination of ϵ -DFA's for grammars 1 and 2 c. ϵ -DFA after subset construction

correspond to ϵ -transitions of the ϵ -NFA. The ϵ -DFA does not contain the ϵ -transition that would result in a self-edge on the station state E . Intuitively, an ϵ -transition from q_0 to q_1 expresses that station state q_1 is supposed to be closed in q_0 (i.e. the subset q_0 is a superset of the subset q_1) as long as the automaton is not changed, which makes self-edges useless since every state is closed in itself. The composition algorithm is oblivious to the set of items that resulted in a DFA state, therefore the states of the automaton no longer contain LR item sets.

Figure 10b combines the ϵ -DFA of 10a with a second ϵ -DFA to form an automaton where the ϵ -transitions become relevant, thus being an ϵ -NFA. The parse table component adds variables to the tiny expression language of grammar 1 based on the following grammar and its ϵ -DFA.

$$\Sigma = \{\text{Id}\} \quad N = \{T\} \quad P = \{T \rightarrow \text{Id}\} \quad (2) \quad T \rightarrow r_8 \xrightarrow{\text{Id}} r_9$$

The combined automaton connects station states of the same nonterminal originating from different parse table components by ϵ -transitions, in this case the two station states q_6 and r_8 for T (bold). Intuitively, these transitions express that the two states should always be part of ϵ -closures (subsets) together. In a combination of the original ϵ -NFA's, the station state T would have ϵ -transitions to all the initial items that now constitute the station states q_6 and r_8 .

Figure 10c is the result of applying subset *reconstruction* to Figure 10b, resulting in a deterministic automaton (ignoring the now irrelevant ϵ -edges). State q_1 is extended to s_1 by including r_8 because there is a new path from q_1 to r_8 over ϵ -transitions, i.e. r_8 enters the ϵ -closure of q_1 . As a result of this extended subset, s_1 now has a transition on Id to s_8 . Similarly, state q_3 is extended to s_3 . Finally, station states q_6 and r_8 are merged into the single state s_6 because of the cycle of ϵ -transitions. Observe that five of the nine states from Figure 10b are not affected because their ϵ -closures have not changed.

4.1 Generating LR(0) Parse Tables Components

The visualizations of automata only show the states, station states and the transitions. However, LR parse tables also have reduce and accept actions and distinguish transitions over terminals (shift actions) from nonterminals (gotos). To completely capture parse tables, we define an LR(0) parse table component T to be a tuple

$$\langle Q, \Sigma, N, \delta, \delta^\epsilon, \text{station}, \text{predict}, \text{reduce}, P, \text{accept} \rangle$$

with Q a set of states, Σ a set of terminal symbols, N a set of nonterminal symbols, δ a transition function $Q \times (\Sigma \cup N) \rightarrow Q$, δ^ϵ a transition function $Q \rightarrow \mathcal{P}(Q)$ (visualized by dashed edges), *station* the function $N \rightarrow Q$ (visualized using arrows into the automaton labelled with a nonterminal), *predict* a function $Q \rightarrow \mathcal{P}(N)$, *reduce* a function $Q \rightarrow \mathcal{P}(P)$, P a set of productions of the form $A \rightarrow \alpha$, and finally $\text{accept} \subseteq Q$.

Note that the δ function of a component returns a single state for a symbol, hence it corresponds to a deterministic automaton. The ϵ -transitions are captured in a separate function δ^ϵ . For notational convenience we do not explicitly restrict the range of the δ^ϵ function to station states in this definition.

Parse table components do not have a specific start nonterminal, instead the *station* function is used to map all nonterminals to a station state. Given the start nonterminal,

the **station** function returns the start state. We say that a station state q is *closed* in a state q' if $q' \rightarrow_\epsilon q$.

Figure 11 shows the algorithm for generating a parse table component, which is very similar to the subset construction algorithm of Figure 9. First, an ϵ -NFA is generated¹ for the grammar G . For every nonterminal A the ϵ -closure (defined in Figure 9) of its station state is added to the set of states Q_D ² of the ϵ -DFA, thus capturing the closure of the initial items of all productions of A .

Next, for every state, the nonterminals⁸ predicted by the items of this subset are determined. The predicted nonterminals correspond to the ϵ -transitions to station states, or equivalently the transitions on nonterminal symbols from this state. The set of predicted symbols (**predict**) is not necessary for a naive implementation of composition, but it will improve the performance as discussed later. The ϵ -transitions⁹ are determined based on the predicted nonterminals, but it could also be based on δ_E . The self-edges are removed by subtracting the state itself. To drive the construction of the DFA, the next states¹⁰ are determined by following the transitions of the ϵ -NFA using the **move** function for all labels of this subset (see Figure 9). Finally, the **reduce** actions¹⁴ of a state are the productions for which there is an item with the dot at the last position. If there is an item that predicts the special **eof** terminal¹⁵, then the state becomes an accepting state. This definition requires the items of a subset to be known to determine accepting states and reduce actions, but this can easily be avoided by extending the ϵ -NFA with reduce actions and accepting states.

```

function generate-xtbl( $G$ ) =
1   $\langle Q_E, \Sigma, \delta_E \rangle := \text{generate-nfa}(G)$ 
2  for each  $A \in N(G)$ 
3     $S := \epsilon\text{-closure}(\{\bullet A\}, \delta_E)$ 
4     $Q_D := Q_D \cup \{S\}$ 
5    station( $A$ ) :=  $S$ 
6  repeat until  $Q_D$  and  $\delta_D$  do not change
7    for each  $S \in Q$ 
8       $\text{predict}(S) := \{A \mid q \in S, q \rightarrow_\epsilon \{\bullet A\} \in \delta_E\}$ 
9       $\delta_D^\epsilon(S) := \{\text{station}(A) \mid A \in \text{predict}(S)\} - \{S\}$ 
10     for each  $X \in \text{labels}(S, \delta_E)$ 
11        $S' := \epsilon\text{-closure}(\text{move}(S, X, \delta_E), \delta_E)$ 
12        $Q_D := Q_D \cup \{S'\}$ 
13        $\delta_D := \delta_D \cup \{S \rightarrow_X S'\}$ 
14      $\text{reduce}(S) := \{A \rightarrow \alpha \mid [A \rightarrow \alpha \bullet] \in S\}$ 
15     if  $[A \rightarrow \alpha \bullet \text{eof}] \in S$  then
16        $\text{accept} := \text{accept} \cup \{S\}$ 
17  return  $\langle Q_D, \Sigma(G), N(G), \delta_D, \delta_D^\epsilon,$ 
18     $\text{station}, \text{predict}, \text{reduce}, P(G), \text{accept} \rangle$ 

```

Fig. 11. LR(0) parse table component generation

4.2 Composing LR(0) Parse Table Components

We first present a high-level version of the composition algorithm that does not take much advantage of the subset construction that has been applied to the individual parse table components. The algorithm is not intended to be implemented in this way, similar to the algorithms for parse table generation. In all cases the fixpoint approach is very inefficient and needs to be replaced by a worklist algorithm. Also, efficient data structures need to be chosen to represent subsets and transitions. Figure 12 shows the high-level algorithm for parse table composition. Again, the algorithm is a variation of subset construction. First, the **combine-xtbl** function is invoked to combine the components (resulting in Figure 10b of the example). The δ^ϵ functions of the individual components

```

function compose-xtbl( $T_0, \dots, T_k$ ) =
1   $\langle N_E, \delta_E, \delta_E^{\epsilon+}, \text{stations}_E, \text{predict}_E, \text{reduce}_E, \text{accept}_E \rangle := \text{combine-xtbl}(T_0, \dots, T_k)$ 
2  for each  $A \in N_E$ 
3     $S := \epsilon\text{-closure}(\text{stations}_E(A), \delta_E^{\epsilon+})$ 
4     $Q_D := Q_D \cup \{S\}$ 
5     $\text{station}(A) := S$ 
6  repeat until  $Q_D$  and  $\delta_D$  do not change
7    for each  $S \in Q_D$ 
8       $\text{predict}(S) := \bigcup \{\text{predict}_E(q) \mid q \in S\}$ 
9       $\delta_D^{\epsilon}(S) := \{\text{station}(A) \mid A \in \text{predict}(S)\} - \{S\}$ 
10     for each  $X \in \text{labels}(S, \delta_E)$ 
11        $S' := \epsilon\text{-closure}(\text{move}(S, X, \delta_E), \delta_E^{\epsilon+})$ 
12        $Q_D := Q_D \cup \{S'\}$ 
13        $\delta_D := \delta_D \cup \{S \rightarrow_X S'\}$ 
14      $\text{reduce}(S) := \bigcup \{\text{reduce}_E(q) \mid q \in S\}$ 
15     if  $(\text{accept}_E \cap S) \neq \emptyset$  then  $\text{accept} := \text{accept} \cup \{S\}$ 
16 return  $\langle Q_D, \bigcup_{i=0}^k \Sigma_i, N_E, \delta_D, \delta_D^{\epsilon}, \text{station}, \text{predict}, \text{reduce}, \bigcup_{i=0}^k P_i, \text{accept} \rangle$ 

function combine-xtbl( $T_0, \dots, T_k$ ) =
1   $N_E := \bigcup_{i=0}^k N(T_i), \quad \delta_E^{\epsilon+} := \bigcup_{i=0}^k \delta^{\epsilon}(T_i), \quad \delta_E := \bigcup_{i=0}^k \delta(T_i)$ 
2  for each  $A \in N_E$ 
3    for  $0 \leq i \leq k$ 
4      for  $0 \leq j \leq k, j \neq i$ 
5        if  $A \in N(T_i) \wedge A \in N(T_j)$ 
6           $\delta_E^{\epsilon+} := \delta_E^{\epsilon+} \cup \{\text{station}(T_i, A) \rightarrow \text{station}(T_j, A)\}$ 
7   $\text{stations}_E := \bigcup_{i=0}^k \text{station}(T_i), \quad \text{predict}_E := \bigcup_{i=0}^k \text{predict}(T_i)$ 
8   $\text{reduce}_E := \bigcup_{i=0}^k \text{reduce}(T_i), \quad \text{accept}_E := \bigcup_{i=0}^k \text{accept}(T_i)$ 
9  return  $\langle N_E, \delta_E, \delta_E^{\epsilon+}, \text{stations}_E, \text{predict}_E, \text{reduce}_E, \text{accept}_E \rangle$ 

```

Fig. 12. LR(0) parse table component composition

are collected into a transition function $\delta_E^{\epsilon+} \stackrel{1}{=}$. To merge the station states $\delta_E^{\epsilon+}$ is extended to connect the station states of the same nonterminal in different components $\stackrel{2}{=}$. Finally, the relations station , predict , and reduce , and the set accept are combined. The domain of the relations predict and reduce are states, which are unique in the combined automaton. Thus, the combined functions predict_E and reduce_E have the same type signature as the functions of the individual components. However, the domain of station functions for individual components might overlap, hence the new function stations_E is a function of $N \rightarrow \mathcal{P}(Q)$.

Back to the compose-xtbl function, we now initialize the subset reconstruction by creating the station states $\stackrel{2}{=}$ of the new parse table. The new station states are determined for every nonterminal by taking the ϵ -closure of the station states of all the components (stations_E) over $\delta_E^{\epsilon+}$. The creation of the station states initializes the fixpoint operation on Q_D and δ_D $\stackrel{6}{=}$. The fixpoint loop is very similar to the fixpoint loop of parse table component generation (Figure 11). If the table is going to be subject to further composition, then the predict $\stackrel{8}{=}$ and δ_D^{ϵ} $\stackrel{9}{=}$ functions can be computed similar to the generation of components. For final parse tables this is not necessary. Next, the transitions to other

states are determined using the move function ¹¹ and the transition function δ_E . Similar to plain LR(0) parse table generation the result of the move function is called a *kernel* (but it is a set of states, not a set of items). The kernel is turned into an ϵ -closure using the extended set of ϵ -transitions, i.e. $\delta_E^{\epsilon+}$. Finally, the reduce actions are simply collected ¹⁴ and if any of the involved states is an accept state, then the composed state will be an accept state ¹⁵.

This algorithm performs complete subset reconstruction, since it does not take into account that many station states are already closed in subsets. Also, it does not use the set of predicted nonterminals in any way. The correctness of the algorithm is easy to see by comparison to the ϵ -NFA approach to LR(0) parser generation. Subset construction can be applied partially to an automaton, so extending a deterministic automaton with new states and transitions and applying subset construction subsequently is not different from applying subset construction to an extension of the original ϵ -NFA.

4.3 Optimization

In the worst case, subset construction applied to a NFA can result in an exponential number of states in the resulting DFA. There is nothing that can be done about the number of states that have to be created in subset reconstruction, except for creating these states as efficiently as possible. As stated by research on subset construction [28,29], it is important to choose the appropriate algorithms and data structures. For example, the fixpoint iteration should be replaced, checking for the existence of a subset of states in Q must be efficient (we use uniquely represented treaps [30]), and the kernel for a transition on a symbol X from a subset must be determined efficiently. In our implementation we have applied some of the basic optimizations, but have focused on optimizations specific to parse table composition. The performance of parse table composition mostly depends on (1) the number of ϵ -closure invocations and (2) the cardinality of the resulting ϵ -closures.

Avoiding Closure Calls. In the plain subset construction algorithm ϵ -closure calls are inevitable for every subset. However, subset construction has already been applied to the parse table components. If we know in advance that for a given kernel a closure call will not add any station states to the closure that are not already closed in the states of the kernel, then we can omit the ϵ -closure call. For the kernel $\text{move}(S, X, \delta_E)$ ¹¹ it is not necessary to compute the ϵ -closure if none of the states in the kernel predict a nonterminal that occurs in more than one parse table component, called an *overlapping* nonterminal. If predicted nonterminals are not overlapping, then the ϵ -transitions from the states in this kernel only refer to station states that have no new ϵ -transitions added by combine_xtbl ². Hence, the ϵ -closure of the kernel would only add station states that are already closed in this kernel. Note that new station states cannot be found indirectly through ϵ -transitions either, because $\forall q_0 \rightarrow q_1 \in \delta^\epsilon(T_i) : \text{predict}(q_0) \supseteq \text{predict}(q_1)$. Thus, the kernel would have predicted the nonterminal of this station state as well.

To prevent unintentional overlap of nonterminals, it is useful to support external and internal symbols. Internal symbols are not visible to other components.

Reduce State Rewriting. If a closure $\epsilon\text{-closure}(\text{move}(S, X, \delta_E), \delta_E^{\epsilon+})$ is a singleton set $\{q_0\}$, then q_0 can be added directly to the states of the composed parse table without any updating of its actions and transitions. That is, not only does the closure $\{q_0\}$ have the same

actions and transitions as q_0 , but the transitions can also use the same names for the states they refer to. However, for this we need to choose the names of new states strategically. If there is a transition $q_0 \rightarrow_X q_1$ in the component of q_0 , then $\text{move}(\{q_0\}, X, \delta_E)$ will always be the single-state kernel $\{q_1\}$, but $\epsilon\text{-closure}(\{q_1\}, \delta_E^{\epsilon+})$ is not necessarily the single-state closure $\{q_1\}$. Hence, it is not straightforward that the transition $q_0 \rightarrow_X q_1$ can be included as is in the composed parse table. To avoid the need for updating transitions from single-state closures, we choose the names for the ϵ -closure of a single-state kernel $\{q\}$ strategically as the name of q .

This optimization makes it very useful to restrict the number of states in a closure aggressively. If the closure is restricted to a single state, then the compose algorithm only needs to traverse the transitions to continue composing states.

Reduce Closure Cardinality. Even if a kernel predicts one or more overlapping symbols (thus requiring an ϵ -closure call) it is not necessarily the case that any station states will be added to the kernel to form a closure. For example, if two components T_0 and T_1 having an overlapping symbol E are composed and two states $q_0 \in Q(T_0)$ and $q_1 \in Q(T_1)$ predicting E are both in a kernel, then the two station states for E are already closed in this kernel. To get an efficient ϵ -closure implementation the closures could be pre-computed *per state* using a transitive closure algorithm, but this example illustrates that the *subset* in which a state occurs will make many station states unnecessary. Note that for a state q in table T not all station states of T are irrelevant. Station states of T might become reachable through ϵ -transitions by a path through a different component.

A first approximation of the station states that need to be added to a kernel S to form a closure is the set of states that enter the ϵ -closure because of the ϵ -transitions added by `combine-xtbl` ²

$$S^{\text{approx}} = \epsilon\text{-closure}(S, \delta_E^{\epsilon+}) - \epsilon\text{-closure}(S, \bigcup \delta_i^{\epsilon})$$

However, another complication compared to an ordinary transitive closure is that the station states have been transitively closed already inside their components. Therefore, we are not interested in *all* states that enter the ϵ -closure, since many station states in S^{approx} are already closed in other station states of S^{approx} . Thus, the minimal set of states that need to be added to a kernel to form a closure is the set of station states that (I) are not closed in the states of the kernel (S^{approx}) and are not already closed by another state in S^{approx} :

$$S^{\text{min}} = \{q_0 \mid q_0 \in S^{\text{approx}}, \nexists q_1 \in S^{\text{approx}} : q_1 \rightarrow q_0 \in \bigcup \delta_i^{\epsilon}\}$$

The essence of the problem is expressed by the *predict graph*, which is a subgraph of the combined graph shown in Figure 10b restricted to station states and ϵ -transitions between them. Every δ_i^{ϵ} transition function induces an acyclic, transitively closed graph, but these two properties are destroyed in the graph induced by $\delta_E^{\epsilon+}$. We distinguish the existing ϵ -transitions and the new transitions introduced by `combine-xtbl` ² in *intra-component* and *inter-component* edges, respectively.

Figure 13 shows the optimized ϵ -closure algorithm, which is based on a traversal of the predict graph. For a given kernel S , the procedure `mark` first marks the states that are already closed in the kernel. If a state q_1 in the subset is a station state, then the station state itself ³ and all the other reachable station states in the same component are

marked⁴. Note that the graph induced by ϵ -transitions of a component is transitively closed, thus there are direct edges to all the reachable station states (intra-edges). For a state q_1 of the subset S that is not a station state, we mark all the station states that are closed in this non-station state q_1 ⁶. Note that q_1 itself is not present in the predict graph. The station states are marked with a source state, rather than a color, to indicate which state is responsible for collecting states in this part of the graph. If the later traversal of the predict graph encounters a state with a source different from the one that initiated the current traversal, then the traversal is stopped. The source marker is used to make the intuition of the algorithm more clear, i.e. a coloring scheme could be used as well.

Next, the function ϵ -closure initiates traversals of the predict graph by invoking the visit function for all the involved station states. The visit function traverses the predict graph, collecting¹⁰ only the states reached *directly* through inter-component edges, thus avoiding station states that are already closed in an earlier discovered station state. For every state, the intra-component edges are visited first, and mark states BLACK as already visited. The visit function stops traversing the graph if it encounters a node with a different source (but continues if the source is NULL), thus avoiding states that are already closed in other station states.

```

procedure visit( $v, src$ )
1  color[ $v$ ] := BLACK
2  result[ $v$ ] :=  $\emptyset$ 
3  for each  $w \in \text{intra-edges}[v]$ 
4    if (source[ $w$ ] =  $src \vee \text{source}[w] = \text{NULL} \wedge \text{color}[w] = \text{WHITE}$ )
5      visit( $w, src$ )
6      result[ $v$ ] := result[ $v$ ]  $\cup$  result[ $w$ ]
7  for each  $w \in \text{inter-edges}[v]$ 
8    if source[ $w$ ] = NULL  $\wedge$  color[ $w$ ] = WHITE
9      visit( $w, src$ )
10     result[ $v$ ] := { $v$ }  $\cup$  result[ $v$ ]  $\cup$  result[ $w$ ]

procedure mark( $S, \delta^\epsilon$ )
1  for each  $q_1 \in S$ 
2    if  $q_1$  is station state
3      source[ $q_1$ ] :=  $q_1$ 
4      for each  $q_2 \in \text{intra-edges}[q_1]$  do source[ $q_2$ ] :=  $q_1$ 
5    else
6      for each  $q_2 \in \delta^\epsilon(q_1)$ 
7        source[ $q_2$ ] :=  $q_2$ 
8        for each  $q_3 \in \text{intra-edges}[q_2]$  do source[ $q_3$ ] :=  $q_2$ 

function  $\epsilon\text{-closure}(S, \delta^\epsilon)$  =
1  color[*] := WHITE, result[*] := NULL, source[*] := NULL
2  result :=  $S$ 
3  mark( $S, \delta^\epsilon$ )
4  for each  $q \in S$ 
5    if  $q$  is station state then maybe-visit( $q$ )
6    else for each  $q_A \in \delta^\epsilon(q)$  do maybe-visit( $q_A$ )
7  return result

8  local procedure maybe-visit( $q$ ) =
9    if source[ $q$ ] =  $q \wedge \text{color}[q] = \text{WHITE}$ 
10     visit( $q, q$ )
11     result := result  $\cup$  result[ $q$ ]

```

Fig. 13. Optimized ϵ -closure implementation

5 Evaluation

In the worst case scenario an LR(0) automaton can change drastically if it is combined with another LR(0) automaton. The composition with an automaton for just a *single* production can introduce an exponential number of new states [20]. Parse table composition cannot circumvent this worst case. Fortunately, grammars of typical programming languages do not exhibit this behaviour. To evaluate our method, we measured how parse table composition performs in *typical* applications. Parse table components usually correspond to languages that form a natural sublanguage. These do not change the structure of the language invasively but hook into the base language at some points.

			sdf pgen		parse table composition											
			productions	symbols	normalization (ms)	table generation (ms)	overlapping symbols	Σ states	composed states	% single state	Σ normalization (ms)	Σ generate-xtbl (ms)	compose-xtbl (ms)	nullables (ms)	first sets (ms)	follow sets (ms)
Str.+XML	782	436	430	230	4	2983	2127	85	160	580	27	0	0	10	10	37
Str.+Java	1766	836	3330	1790	3	6513	6612	93	2110	4250	67	0	0	10	20	80
Str.+Stratego	1115	536	780	600	4	4822	4085	89	410	1530	37	0	0	7	7	47
Str.+Stratego+XML	1420	745	1530	830	5	5752	4807	89	440	1600	60	0	3	10	13	77
Str.+Stratego+Java	2404	1145	6180	2710	4	10405	9295	93	2390	5780	127	0	3	20	23	150
Java+SQL	1391	750	2800	1300	3	5175	3698	92	1350	2440	63	0	0	10	10	73
Java+XPath	1084	554	1560	780	3	4158	2848	90	1150	2010	60	0	0	3	3	63
Java+LDAP	1024	545	1550	760	3	3831	2467	89	1140	1940	43	0	0	3	3	53
Java+XPath+SQL	1580	853	3560	1290	3	5784	4272	93	1390	2530	63	0	0	10	13	83
Java+XPath+LDAP	1213	648	1910	1050	3	4440	3041	91	1170	2030	57	0	3	3	10	63
AspectJ + 5x Java	3388	2426	48759	10261	62	19332	8305	51	1460	1990	477	0	3	30	33	520

Fig. 14. Benchmark of parse table composition compared to the SDF parser generator. (1) number of reachable productions and (2) symbols, (3) time for normalizing the full grammar and (4) generating a parse table using SDF pgen, (5) number of overlapping symbols, (6) total number of states in the components, (7) number of states after composition, (8) percentage of single-state closures, (9) total time for normalizing the individual components and (10) generating parse table components, (11) time for reconstructing the LR(0) automaton, (12, 13, 14, 15) time for various aspects of follow sets, (16) total composition time (11 + 15). We measure time using the clock function, which reports time in units of 10ms on our machine. We measure total time separately, which explains why some numbers do not sum up exactly. Results are the average of three runs.

We compare the performance of our prototype to the SDF parser generator that targets the same scannerless GLR parser (sdf2-bundle-2.4pre212034). SDF grammars are compiled by first normalizing the high-level features to a core language and next applying the parser generator. Our parser generator accepts the same core language as input. The prototype consists of two main tools: one for generating parse table components and one for composing them. As opposed to the SDF parser generator, the generator of our prototype has not been heavily optimized because its performance is not relevant to the performance of runtime parse table composition. In our comparison, we assume that the performance of the SDF parser generator is typical for a parser generator, i.e. it has no unusual performance deficiencies. This is a risk in our evaluation, and a more extensive evaluation to other parser generators would be valuable. We have implemented a generator and composer for *scannerless* GLR SLR parse tables. The efficient algorithms for the SLR and scannerless extensions of the LR(0) algorithm are presented in [31] (Section 6.6 and 6.7). Scannerless parsing affects the performance of the composer: grammars have more productions, more symbols, and layout occurs between many symbols. The composed parse tables are identical to parse tables generated using the SDF parser generator. Therefore, the performance of the parsers is equivalent.

Figure 14 presents the results for a series of metaprogramming concrete syntax extensions using Stratego [10], StringBorg [5] extensions, and AspectJ. For Stratego and StringBorg, the number of overlapping symbols is very limited and there are many single-state closures. Depending on the application, different comparisons of the timings are useful. For runtime parse table composition, we need to compare the performance to generation of the full automaton. The total composition time (col. 16) is only about 2% to 16% of the SDF parse table generation time (col. 4). The performance benefit increases more if we include the required normalization (col. 3) (SDF does not support separate normalization of modules). For larger grammars (e.g. involving Java) the performance benefit is bigger.

The AspectJ composition is clearly different from the other embeddings. The AspectJ grammar [32] uses grammar mixins to reuse the Java grammar in 5 different contexts. All contexts customize their instance of Java, e.g. reserved keywords differ per context. Without separate compilation this results in 5 copies of the Java grammar, which all need to be compiled, thus exercising the SDF parser generator heavily. With separate compilation, the Java grammar can be compiled once, thus avoiding a lot of redundant work. This composition is not intended to be used at runtime, but serves as an example that separate compilation of grammars is a major benefit if multiple instance of the same grammar occur in a composition. The total time to generate a parse table from source (col. 10 + 11 + 16) is only 7% of the time used by the SDF parser generator.

6 Related and Future Work

Modular Grammar Formalisms. There are a number of parser generators that support splitting a grammar into multiple files, e.g. Rats! [33], JTS [9], PPG [4], and SDF [19]. They vary in the expressiveness of their modularity features, their support for the extension of lexical syntax, and the parsing algorithm that is employed. Many tools ignore the intricate lexical aspects of syntax extensions, whereas some apply scannerless parsing or context-aware scanning [34]. However, except for a few research prototypes discussed next, these parser generators all generate a parser by first collecting all the sources, essentially resulting in whole-program compilation.

Extensible Parsing Algorithms. For almost every single parsing algorithm extensible variants have already been proposed. What distinguishes our work from existing work is the idea of separately compiled parse table components and a solid foundation on finite automata for combining these parse table components. The close relation of our principles to the LR parser generation algorithm makes our method easy to comprehend and optimize. All other methods focus on adding productions to an existing parse table, motivated by applications such as interactive grammar development. However, for the application in extensible compilers we do not need incremental but *compositional* parser generation. In this way, the syntax of a language extension can be compiled, checked, and deployed independently of the base language in which it will be used. Next, we discuss a few of the related approaches.

Horspool's [20] method for incremental generation of LR parsers is most related to our parse table composition method. Horspool presents methods for adding and deleting productions from LR(0), SLR, as well as LALR(1) parse tables. The work

is motivated by the need for efficient grammar debugging and interactive grammar development, where it is natural to focus on addition and deletion of productions instead of parse table components. Interactive grammar development requires the (possibly incomplete) grammar to be able to parse inputs all the time, which somewhat complicates the method. For SLR follow sets Horspool uses an incremental transitive closure algorithm based on a matrix representation of the first and follow relations. In our experience, the matrix is very sparse, therefore we use Digraph. This could be done incrementally as well, but due to the very limited amount of time spend on the follow sets, it is hard to make a substantial difference.

IPG [21] is a lazy and incremental parser generator targeting a GLR parser using LR(0) parse tables. This work was motivated by interactive metaprogramming environments. The parse table is generated by need during the parsing process. IPG can deal with modifications of the grammar as a result of the addition or deletion of rules by resetting states so that they will be reconstructed using the lazy parser generator. Rek-ers [18] also proposed a method for generating a single parse table for a set of languages and restricting the parse table for parsing specific languages. This method is not applicable to our applications, since the syntax extensions are not a fixed set and typically provided by other parties.

Dypgen [15] is a GLR self-extensible parser generator focusing on scoped modification of the grammar from its semantic actions. On modification of the grammar it generates a new LR(0) automaton. Dypgen is currently being extended by its developers to incorporate our algorithm for runtime extensibility. *Earley* [35] parsers work directly on the productions of a context-free grammar at parse-time. Because of this the Earley algorithm is relatively easy to extend to an extensible parser [36,37]. Due to the lack of a generation phase, Earley parsers are less efficient than GLR parsers for programming languages that are close to LR. *Maya* [38] uses LALR for providing extensible syntax but regenerates the automaton from scratch for every extension. *Cardelli's* [22] extensible syntax uses an extensible LL(1) parser. *Camlp4* [39] is a preprocessor for OCaml using an extensible top down recursive descent parser.

Tatoo [40] allows incomplete grammars to be compiled into separate parse tables that can be linked together. However, Tatoo does not compose parse tables, but parsers. It switches to a different parser when it encounters an error. This alternative approach has disadvantages: (1) changes from parser to parser have to be uniquely identified by an unexpected token; (2) grammar developers have to be careful that errors are triggered at locations where a switch to a different parser is required; and (3) the result of linking two separately compiled grammars does not define the same language as the composed grammar. In Tatoo, the linking between grammar modules is defined more explicitly. It is not possible to define productions for the same nonterminal in different modules. Thus, composition is more coarse-grained than parse table composition. This makes the error-based approach more effective than in a fine-grained approach.

Automata Theory and Applications. The *egrep* pattern matching tool uses a DFA for efficient matching in combination with lazy state construction to avoid the initial overhead of constructing a DFA. *egrep* determines the transitions of the DFA only when they are actually needed at runtime. Conceptually, this is related to lazy parse table construction in IPG. It might be an interesting experiment to apply our subset reconstruction in such

a lazy way. Essentially, parse table composition is a DFA maintenance problem. Surprisingly, while there has been a lot of work in the maintenance of transitive closures, we have not been able to find existing work on DFA maintenance.

Future Work. We presented the core algorithm for parse table composition. We plan to extend the algorithm to support features that grammar formalisms typically add to basic context-free grammars, such as precedence declarations. Currently, precedence declarations across parse table components are not supported by our algorithm. Other open issues are designing a module system for a grammar formalism that takes online composition of parse table components into account. In particular, there is a need for defining the interface of a parse table component. Our current prototype supports the basic primitives, such as external and internal symbols, but policies are necessary on top of this. Finally, it would be interesting to integrate our algorithm in an LR parser generator that generates executable code. Usually, these parser generators output a parse table together with the engine for interpreting the parse table. Therefore, with a bit of additional bookkeeping our algorithm can still be applied in this setting.

References

1. Van Wyk, E., Krishnan, L., Bodin, D., Schwerdfeger, A.: Attribute grammar-based language extensions for java. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 575–599. Springer, Heidelberg (2007)
2. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. In: Proc. of the Seventh Workshop on Language Descriptions, Tools and Applications (LDTA 2007). ENTCS, vol. 203, pp. 103–116. Elsevier, Amsterdam (2008)
3. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: OOPSLA 2007: Proc. of the 22nd ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, pp. 1–18. ACM, New York (2007)
4. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for java. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
5. Bravenboer, M., Dolstra, E., Visser, E.: Preventing injection attacks with syntax embedding – a host and guest language independent approach. In: GPCE 2007: Proc. of the 6th Intl. Conf. on Generative Programming and Component Engineering, pp. 3–12. ACM, New York (2007)
6. Lhoták, O., Hendren, L.: Jedd: A BDD-based relational extension of Java. In: Proc. of the ACM SIGPLAN 2004 Conf. on Programming Language Design and Implementation (2004)
7. Millstein, T.: Practical predicate dispatch. In: OOPSLA 2004: Proc. of Conf. on Object Oriented Programming, Systems, Languages, and Applications, pp. 345–364. ACM, New York (2004)
8. Arnoldus, B.J., Bijpost, J.W., van den Brand, M.G.J.: Repleo: A syntax-safe template engine. In: GPCE 2007: Proc. of the 6th Intl. Conf. on Generative Programming and Component Engineering, pp. 25–32. ACM, New York (2007)
9. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: tools for implementing domain-specific languages. In: Proc. Fifth Intl. Conf. on Software Reuse (ICSR 1998), pp. 143–153. IEEE Computer Society Press, Los Alamitos (1998)
10. Visser, E.: Meta-programming with concrete object syntax. In: Batory, D., Consel, C., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 299–315. Springer, Heidelberg (2002)

11. ASF+SDF Meta-Environment website, <http://www.meta-environment.org>
12. van Wyk, E., Bodin, D., Huntington, P.: Adding syntax and static analysis to libraries via extensible compilers and language extensions. In: Proc. of Library-Centric Software Design (LCSD 2006), pp. 35–44 (2006)
13. Nystrom, N., Qi, X., Myers, A.C.: J&: nested intersection for scalable software composition. In: OOPSLA 2006: Proc. of Conf. on Object Oriented Programming Systems, Languages, and Applications, pp. 21–36. ACM, New York (2006)
14. Odersky, M., Zenger, M.: Scalable component abstractions. In: OOPSLA 2005: Proc. of Conf. on Object Oriented Programming, Systems, Languages, and Applications, pp. 41–57. ACM, New York (2005)
15. Onzon, E.: Dypgen: Self-extensible parsers for ocaml (2007), <http://dypgen.free.fr>
16. Salomon, D.J., Cormack, G.V.: Scannerless NSLR(1) parsing of programming languages. In: PLDI 1989: Proc. of the ACM SIGPLAN 1989 Conf. on Programming Language Design and Implementation, pp. 170–178. ACM, New York (1989)
17. Tomita, M.: Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Dordrecht (1985)
18. Rekers, J.: Parser Generation for Interactive Environments. PhD thesis, University of Amsterdam (1992)
19. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (September 1997)
20. Horspool, R.N.: Incremental generation of LR parsers. *Computer Languages* 15(4), 205–223 (1990)
21. Heering, J., Klint, P., Rekers, J.: Incremental generation of parsers. *IEEE Transactions on Software Engineering* 16(12), 1344–1351 (1990)
22. Cardelli, L., Matthes, F., Abadi, M.: Extensible syntax with lexical scoping. SRC Research Report 121, Digital Systems Research Center, Palo Alto, California (February 1994)
23. Knuth, D.E.: On the translation of languages from left to right. *Information and Control* 8(6), 607–639 (1965)
24. Aho, A.V., Sethi, R., Ullman, J.: Compilers: Principles, techniques, and tools. Addison Wesley, Reading (1986)
25. Grune, D., Jacobs, C.J.H.: Parsing Techniques - A Practical Guide. Ellis Horwood, Upper Saddle River (1990)
26. Johnstone, A., Scott, E.: Generalised reduction modified LR parsing for domain specific language prototyping. In: 35th Annual Hawaii Intl. Conf. on System Sciences (HICSS 2002), Washington, DC, USA, p. 282. IEEE Computer Society Press, Los Alamitos (2002)
27. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley, Boston (2006)
28. Leslie, T.: Efficient approaches to subset construction. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada (1995)
29. van Noord, G.: Treatment of epsilon moves in subset construction. *Computational Linguistics* 26(1), 61–76 (2000)
30. Seidel, R., Aragon, C.R.: Randomized search trees. *Algorithmica* 16(4/5), 464–497 (1996)
31. Bravenboer, M.: Exercises in Free Syntax. Syntax Definition, Parsing, and Assimilation of Language Conglomerates. PhD thesis, Utrecht University, The Netherlands (January 2008)
32. Bravenboer, M., Tanter, E., Visser, E.: Declarative, formal, and extensible syntax definition for AspectJ – A case for scannerless generalized-LR parsing. In: OOPSLA 2006: Proc. of the 21st ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications, pp. 209–228. ACM, New York (2006)
33. Grimm, R.: Better extensibility through modular syntax. In: PLDI 2006: Proc. of Conf. on Programming Language Design and Implementation, pp. 38–51. ACM, New York (2006)

34. van Wyk, E., Schwerdfeger, A.: Context-aware scanning for parsing extensible languages. In: GPCE 2007: Proc. of the 6th Intl. Conf. on Generative Programming and Component Engineering, pp. 63–72. ACM, New York (2007)
35. Earley, J.: An efficient context-free parsing algorithm. *Communications of the ACM* 13(2), 94–102 (1970)
36. Tratt, L.: Domain specific language implementation via compile-time meta-programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30(6), 1–40 (2008)
37. Kolbly, D.M.: Extensible Language Implementation. PhD thesis, University of Texas at Austin (December 2002)
38. Baker, J., Hsieh, W.: Maya: multiple-dispatch syntax extension in Java. In: PLDI 2002: Proc. of the ACM SIGPLAN 2002 Conf. on Programming Language Design and Implementation, pp. 270–281. ACM, New York (2002)
39. de Rauglaudre, D.: Camlp4 Reference Manual (September 2003)
40. Cervelle, J., Forax, R., Roussel, G.: Separate compilation of grammars with Tatoo. In: Proc. of the Intl. Multiconf. on Computer Science and Information Technology, pp. 1093–1101 (2007)

Practical Scope Recovery Using Bridge Parsing

Emma Nilsson-Nyman¹, Torbjörn Ekman², and Görel Hedin¹

¹ Department of Computer Science, Lund University, Lund, Sweden
(emma.nilsson_nyman|gorel.hedin)@cs.lth.se

² Programming Tools Group, University of Oxford, Oxford, United Kingdom
torbjorn.ekman@comlab.ox.ac.uk

Abstract. Interactive development environments (IDEs) increase programmer productivity, but unfortunately also the burden on language implementors since sophisticated tool support is expected even for small domain-specific languages. Our goal is to alleviate that burden, by generating IDEs from high-level language specifications using the JastAdd meta-compiler system. This puts increased tension on scope recovery in parsers, since at least a partial AST is required by the system to perform static analysis, such as name completion and context sensitive search.

In this paper we present a novel recovery algorithm called bridge parsing, which provides a light-weight recovery mechanism that complements existing parsing recovery techniques. An initial phase recovers nesting structure in source files making them easier to process by existing parsers. This enables batch parser generators with existing grammars to be used in an interactive setting with minor or no modifications.

We have implemented bridge parsing in a generic extensible IDE for JastAdd based compilers. It is independent of parsing technology, which we validate by showing how it improves recovery in a set of typical interactive editing scenarios for three parser generators: ANTLR (LL(variable lookahead) parsers), LPG (LALR(k) parsers), and Beaver (LALR(1) parsers). ANTLR and LPG both contain sophisticated support for error recovery, while Beaver requires manual error productions. Bridge parsing complements these techniques and yields better recovery for all these tools with only minimal changes to existing grammars.

1 Introduction

Interactive development environments (IDE) have become the tool of choice in large-scale software development. This drastically increases the burden on language developers since sophisticated tool support is expected even for small domain-specific languages. The work presented in this paper is part of a larger effort to generate IDEs from high-level language specifications based on attribute grammars in the JastAdd meta-compiler tools. The AST is used as the predominant data structure and all static semantic analyses are implemented as attribute grammars on top of that tree. This approach has been used successfully to implement a production-quality Java compiler [7], extensible refactoring tools [17], source level control-flow and data-flow analyses [15], and various Java extensions [2, 10].

One key insight from our earlier work is that we can use the AST as the only model of the program and then superimpose additional graph structure using attributes on top

of that tree, e.g., name bindings, inheritance hierarchies and call graphs. The IDE can then reuse and extend this model to provide services such as name completion, cross-referencing, code outline, and semantic search facilities. However, this allows us to use the same extension mechanisms that have proven successful in building extensible compilers to build extensible IDEs. This puts extreme tension on the error recovery facilities in parsers since incomplete programs are the norm rather than the exception during interactive development, and a recovered AST is necessary for instant feedback to the user. An unfortunate consequence of this challenge is that, despite the wealth of research in automatic parser generators from high-level grammars and sophisticated error recovery mechanisms, most IDEs still rely on hand crafted parsers to provide such services to the user.

In this paper we present an algorithm for scope recovery, preserving as much of the AST structure as possible, that is neither tied to a particular parsing technology nor to a specific parser generator. A light-weight pre-processor takes an incomplete program and recovers scope nesting before the adjusted source file is fed to a traditional parser. Existing error recovery mechanisms are then sufficient to build a partial AST suitable for static semantic analysis. This approach even makes it feasible to use a batch parser rather than an incremental parser since the speed of parsing a complete source unit is usually satisfactory even for interactive editing on today's hardware.

The approach has proven successful when combined with several parser generators in our IDE generator for JastAdd based compilers in the Eclipse IDE framework. We have integrated bridge parsing in an IDE framework for compilers where all services use the AST as the only model to extract program information from. An IDE based on Eclipse is generated from an attribute grammar for static semantic analysis and our largest example is an environment for Java which includes support for name completion, content outline, cross-referencing, and various semantic search facilities.

We demonstrate that the approach is independent of parsing technology and parser generator by combining bridge parsing with three different parser generators: ANTLR which generates an LL(variable lookahead) parser, LPG which generates an LALR(k) parser, and Beaver which generates an LALR(1) parser. LPG and ANTLR are particularly interesting because of their sophisticated error recovery mechanisms. We show that on a set of typical interactive editing scenarios, bridge parsing improves recovery for both these tools and the few cases with degraded performance can be mitigated by minor enhancements in the IDE. The contributions of this paper are:

- A general algorithm for recovering scope information suitable for arbitrary parsing technologies.
- An implementation in an IDE generator for JastAdd based compilers in Eclipse.
- A careful comparison of its effect on error recovery in state of the art parser generators.

The rest of the paper is structured as follows. Section 2 explains the requirements on error recovery and outlines previous work and room for improvement. Bridge parsing is introduced in Section 3 and an example of language sensitive recovery for Java is presented in Section 4 and evaluated in Section 5. We finally discuss future work and conclude in Section 6.

2 Background

We first outline scenarios requiring error recovery in the setting described above, and then survey existing recovery mechanisms and explain why they are not sufficient or have room for improvement. This serves as a background and related work before we introduce bridge parsing in Section 3. A more thorough survey of error recovery techniques is available in [5].

2.1 Error Recovery Scenarios

The interactive setting this approach is used in puts extreme pressure on error recovery during parsing. Incomplete programs with errors are the norm rather than the exception, and we rely on an AST to be built to perform most kinds of IDE services to the user. It is therefore of paramount importance that common editing scenarios produce an AST rather than a parse failure to allow for services such as name completion while editing. There is also a tension between sophisticated error recovery and the goal to lower the burden on language developers building IDEs. Ideally, she should be able to reuse an existing parser, used in the compiler, in the IDE with minimal changes while providing satisfactory recovery. We define the following desirable properties of error recovery in this context:

- Support for arbitrary parser generators and parsing formalisms.
- Only moderate additions to add recovery to a specific language grammar.
- Effective in that recovery is excellent for common editing scenarios.

The motivation behind the goal of using arbitrary parser generators is that JastAdd defines its own abstract grammar and as long as the parser can build an AST that adheres to that grammar it can be used with JastAdd. We can thus benefit from the wealth of available parsing techniques by selecting the most appropriate for the language at hand. Notice that many proposed parser formalisms are orthogonal to the problem of handling incomplete programs, e.g., GLR-parsing[18], Earley-parsing[6], and Parsing Expression Grammars [8], all deal with ambiguities in grammars rather than errors in programs.

The overall goal of the project is to lower the burden on language developers who want to provide IDE support for their languages. The extra effort to handle incomplete programs should therefore be moderate compared to the overall effort of lexical and syntactic analysis.

The effectiveness criterion is a very pragmatic one. We want the automatic recovery to be as close as possible to what a user would manually do to correct the problem. Here we borrow the taxonomy from Pennello and DeRemer [16] and consider a correction *excellent* if it repairs the text as a human reader would have, otherwise as *good* if the result is a reasonable program and no spurious errors are introduced, and otherwise as *poor* if spurious errors are introduced.

Consider the simple incomplete program below. A class `C` with a method `m()` and a field `x` is currently being edited. There are two closing curly braces missing. An excellent recovery, and the desired result of an automatic recovery, would be to insert a curly brace before and after the field `x`. A simpler recovery, with poor result, would be to insert two curly braces at the end of the program.

```
class C {  
    void m() {  
        int y;  
        int x;
```

Notice that we need to consider indentation to get an excellent result. Changing the indentation of the field `x` to the same as for the variable `y` should, for instance, change its interpretation to a variable and result in a recovery where both closing braces are inserted at the end of the file. Meaning that the simpler recovery alternative above would be sufficient.

2.2 Simple Recovery

The simplest form of recovery is to let the parser automatically replace, remove, or insert single tokens to correct an erroneous program. This information can easily be extracted from the grammar and is supported by many parser generators. However, for incomplete programs this is insufficient since series of consecutive tokens are usually missing in the source file. It should be noted that this kind of recovery serves as a nice complement to other recovery strategies by correcting certain spelling errors.

2.3 Phrase Recovery

A more advanced form of recovery is phrase level recovery where text that precedes, follows, or surrounds an error token is replaced by a suitable nonterminal symbol [9]. Beacon symbols, e.g., delimiters, and keywords, are used to re-synch the currently parsed production and erroneous tokens are removed from the input stream and replaced by a particular error token. These symbols are language specific and the parsing grammar therefore needs to be extended with error productions unless automatically derived by the parser generator. This form of recovery works very well in practice when beacon symbols are available in the input stream and implemented in many parsing tools. Incomplete programs usually lack some beacons required to re-synch the parser and often result in a parsing failure where the recovery can not proceed.

2.4 Scope Recovery

Hierarchical structure is usually represented by nested scopes in programming languages, e.g., blocks and parentheses. It is a common error to forget to close such scopes and during interactive editing many scopes will be incomplete. Burke and Fisher introduced scope recovery to alleviate such problems [3]. Their technique requires the language developer to explicitly provide symbols that are closing scopes. Charles improves on that analysis by automatically deriving such symbols from a grammar by analyzing recursively defined rules [4]. Scope recovery can drastically improve the performance of phrase recovery since symbols to open and close scopes are often used as beacons to re-synch the parser. Scope recovery usually discards indentation information which unfortunately limits the possible result of the analysis to good rather than excellent for the previously outlined example.

2.5 Incremental Parsing

An interactive setting makes it possible to take history into account when detecting and recovering from errors. This opens up for assigning blame to the actual edit that introduced an error, rather than to the location in the code where the error was detected. A source unit is not parsed from scratch upon a change but instead only the changes are analyzed. Wagner and Graham present an integrated approach of incremental parsing and a self versioning document model in [21,20]. It is worth noting that this requires a deep integration of the environment and the generated parser, which makes it less suitable for our setting due to the goal of supporting multiple parser generators.

2.6 Island Parsing

Island parsing is not an error recovery mechanism per se but rather a general technique to create robust parsers that are tolerant to syntactic errors, incomplete source code, and various language dialects [13]. It is based on the observation that for source model extraction one is often only interested in a limited subset of all language features. A grammar consists of detailed productions describing language constructs of interests, that are called islands, and liberal productions matching the remaining constructs, that are called water. This allows erroneous programs to be parsed as long as the errors are contained in water. The parts missing in an incomplete program are usually a combination of both water and islands which makes this approach less suitable for extracting hierarchical structure on its own.

3 Bridge Parsing

We now present *bridge parsing* as a technique to recover scope information from an incomplete or erroneous source file. It combines island parsing with layout sensitive scope recovery. In [13] Moonen defines island grammars [14,19] as follows:

*An **island grammar** is a grammar that consists of two parts: (i) detailed productions that describe the language constructs that we are particularly interested in (so called **islands**), and (ii) liberal productions that catch the remainder of the input (so called **water**).*

In bridge parsing, tokens which open or close scopes are defined as islands while the rest of the source text is defined as water or reefs. Reefs are described further in Section 3.1. This light-weight parsing strategy is used to detect open scopes and to close them. The end product of the bridge parser is a recovered source representation suitable for any parser that supports phrase level recovery.

A bridge parser is built from three parts, as illustrated in Figure 1. The first part, the tokenizer, takes a source text and produces a list of tokens based on definitions in the *bridge grammar*. The second part, the bridge builder, constructs a bridge model from the token list and the last part, the bridge repairer, analyses and repairs the bridge model.

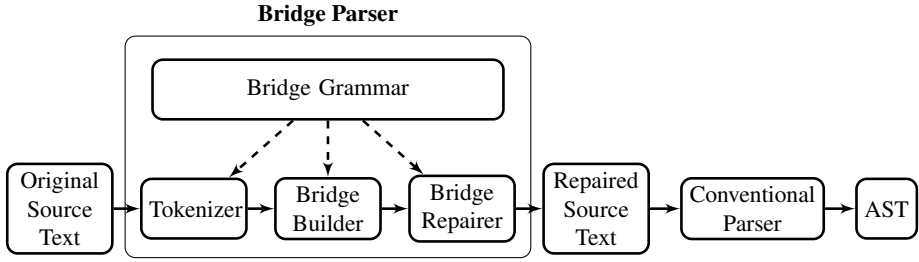


Fig. 1. A bridge parser consists of three parts – a tokenizer returning a token list, a bridge builder taking a token list and returning a bridge model and a bridge repairer which takes a bridge model and generates a repaired source text

3.1 Tokenizer

The goal of the tokenizer is to produce a list of tokens from a source text. Token definitions are provided by a bridge grammar which extends the island grammar formalism with bridges and reefs:

*A **bridge grammar** extends an island grammar with the notions of **bridges** and **reefs**: (i) reefs are attributed tokens which add information to nearby islands, and (ii) bridges connect matching islands. All islands in a bridge grammar are seen as potential bridge abutments.*

We are primarily interested in tokens that define hierarchical structure through nested scopes. Tokens that open or close a scope are therefore represented as islands in the grammar, e.g., braces and parentheses in Java, while most other syntactic constructs are considered water. However, additional tokens in the stream may be interesting to help match two islands that open and close a particular scope. Typical examples include indentation and delimiters, and we call such tokens *reefs*. Reefs can be annotated with attributes which enables comparison between reefs of the same type. Indentation reefs may for instance have different indentation levels. We call such tokens attributed tokens:

***Attributed tokens** are tokens which have attributes, which potentially makes them different from other attributed tokens of the same type. This difference makes it possible to compare attributed tokens to each other.*

The first step when defining a bridge grammar, is to specify islands and reefs. The following example, which we will use as a running example, shows the first part of a bridge grammar relevant to the tokenizer:

Listing 1.1. Tokenizer Definitions

```
1 islands SOF, EOF, A=., B=., C=., D=..
2 reefs R(attr)=..
```

The bridge grammar defines four island types and one reef type. The SOF and EOF islands, represent *start of file* and *end of file*. The A and B islands could, for instance, be open and close brace and C and D open and close parenthesis. The reef could represent indentation where the value of *attr* is the level of indentation. In our implementation each island corresponds to a class that accepts and consumes the substring matching its lexical representation. The information given so far in the example grammar is sufficient to create a lexer which will produce a list of tokens, e.g., the example in Figure 2.

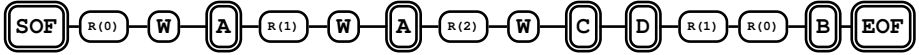


Fig. 2. The token list is a simple double-linked list of *islands* (A, B, C, D), *reefs* (R) and *water* (W). Two additional nodes representing *start of file* (SOF) and *end of file* (EOF) are added first and last in the list. The numbers within parentheses show the attribute values of the reefs.

3.2 Bridge Builder

The bridge builder takes a token list and produces a bridge model defined as follows:

*A **bridge model** is a token list where matched islands are linked with bridges in alignment with the nesting structure in a source text. Bridges can be enclosing other bridges or be enclosed by other bridges, or both, but they never cross each other in an ill-formed manner. Unmatched islands point out broken parts of the nesting structure.*

In the bridge model, islands opening and closing a scope should be linked with a bridge. For the bridge builder to know between which islands to build bridges we need to add definitions to the bridge grammar.

Listing 1.2. Bridge Builder Definitions

```

1 bridge from SOF to EOF { ... }
2 bridge from [a:R A] to [b:R B] when a.attr = b.attr { ... }
3 bridge from [a:R C] to [b:R D] when a.attr = b.attr { ... }

```

The first bridge, the *file bridge*, between the SOF island and EOF island does not need any additional matching constraints i.e., constraints that define when two islands of given types match. For other bridges additional constraints besides type information i.e., an island of type A and type B match, are usually needed. In the example above the islands of type A and B match if both islands have reefs of type R to the left which are equal. The constraints for the third bridge are similar.

The order of the bridge abutments in the definition should correspond to the order in the source text e.g., the bridge start of type C is expected to occur before the bridge end of type D.

With the information given so far in our example bridge grammar we can construct a bridge model. The BRIDGE-BUILDER algorithm will construct as many bridges as possible. If at some point no matching island can be found the algorithm will jump over the unmatched island and eventually construct a bridge enclosing the island. The algorithm is finished when the file bridge has been constructed.

The BRIDGE-BUILDER algorithm, listed in Figure 4, will iterate through the token list until it manages to build the file bridge. For each unmatched bridge start it will try to match it to the next island. If there is a match a bridge will be built and otherwise the algorithm will continue with the next unmatched bridge start. Each time a bridge is encountered it will be crossed. In this way the number of encountered unmatched islands will decrease each iteration.

The algorithm will try to match an unmatched island to the next unmatched island within the current tolerance. The tolerance defines how many unmatched islands that are allowed below a bridge. This value is only changed between iterations and depends on the success of the last iteration. The tolerance should be as low as possible. We start out with a tolerance of zero, meaning no unmatched islands under a bridge. This value will increase if we fail to build bridges during an iteration. If we have a successful iteration we reset the tolerance to zero. If we apply this algorithm to the running example we end up with four iterations, illustrated in Figure 3.

3.3 Bridge Repairer

The goal of the bridge repairer is to analyze the bridge model and repair it if necessary. The BRIDGE-REPAIRER algorithm needs to locate remaining unmatched islands and to add *artificial islands* which will match them. Each island defined in the bridge grammar can potentially be missing. When an island is missing, the algorithm will search for an appropriate *construction site* where an artificial island can be inserted. The search for a construction site starts from the bridge start if the bridge end is missing, and vice versa, and ends when a match is found or an enclosing bridge is encountered. With this in mind we add additional definitions to our example bridge grammar:

Listing 1.3. Bridge Repair Definitions

```

1  bridge from [a:R A] to [b:R B] when a.attr = b.attr {
2      missing[A] find [c:R] where c.attr = b.attr insert after
3      missing[B] find [c:R] where c.attr = a.attr insert after
4  }
5  bridge from [a:R C] to [b:R D] when a.attr = b.attr {
6      missing[C] find [c:R] where c.attr = b.attr insert after
7      missing[D] find [c:R] where c.attr = a.attr insert after
8  }
```

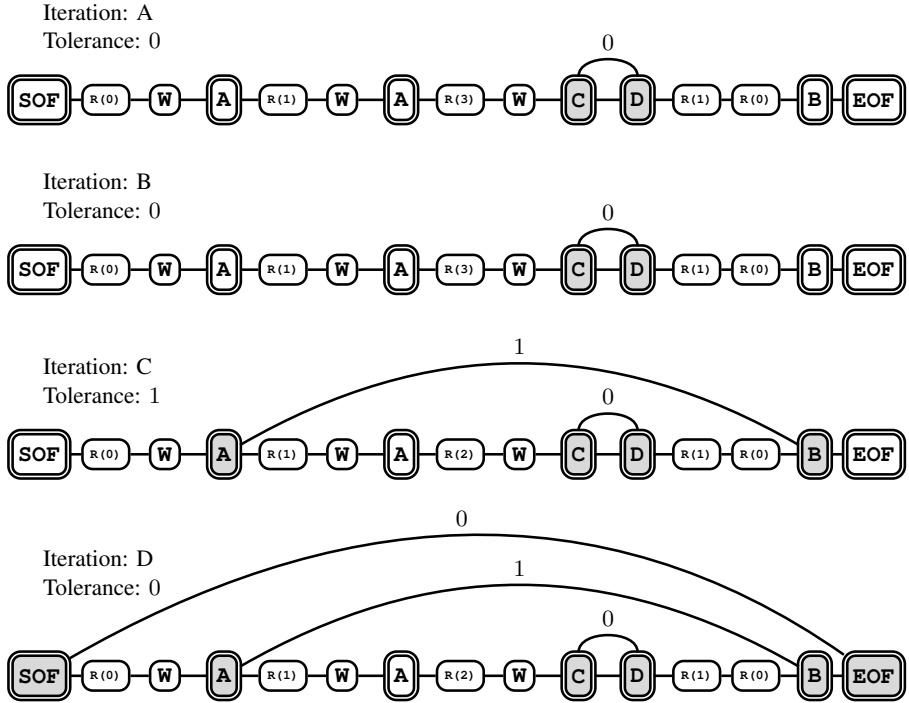


Fig. 3. The resulting bridge model after running the BRIDGE-BUILDER algorithm. No bridges were built during iteration B which results in an increased tolerance in iteration C. During iteration C a bridge is built which means the tolerance is reset to zero in iteration D.

If an island of type A is missing, a construction site will be found in the interval starting at the unmatched island of type B and ending at the start of the enclosing bridge. The first reef of type R which is equal to the reef to the left of the unmatched island of type B points out a construction site. The final information we need is how to insert the artificial island. In this case the artificial island should be inserted after the reef. The definitions for the remaining islands are similar.

The BRIDGE-REPAIRER algorithm, listed in Figure 5, recursively repairs unmatched islands under a bridge, starting with the file bridge. When an unmatched island is encountered the MEND algorithm, listed in Figure 6, will locate a construction site and insert an artificial island.

In the running example, an island of type A is missing an island of type B. The island of type A has a reef of type R on its left hand side with value 1 which means the algorithm will search for for a another reef the same type with the same value. The search is stopped when either a reef is found or a closing island of an enclosing scope is

BUILD-BRIDGES(*sof*)

```

1  tol  $\leftarrow$  0
2  while  $\neg$ HAS-BRIDGE(sof)
3      do start  $\leftarrow$  sof
4          change  $\leftarrow$  FALSE
5          while start  $\neq$  NIL
6              do end  $\leftarrow$  NEXT-UNMATCHED-ISLAND(start, tol)
7                  if BRIDGE-MATCH(start, end)
8                      then BUILD-BRIDGE(start, end)
9                      change  $\leftarrow$  TRUE
10                     start  $\leftarrow$  NEXT-UNMATCHED-START-ISLAND(end)
11                     else start  $\leftarrow$  NEXT-UNMATCHED-START-ISLAND(start)
12         if  $\neg$  change
13             then tol  $\leftarrow$  tol + 1
14             else if tol > 0
15                 then tol  $\leftarrow$  0

```

Fig. 4. The BUILD-BRIDGES algorithm constructs a bridge model from a token list. The NEXT-UNMATCHED-ISLAND returns the next unmatched island to the right of the given island. The tolerance defines the number of unmatched islands to jump over before the procedure returns. NEXT-UNMATCHED-START-ISLAND is similar but only looks for bridge starts.

BRIDGE-REPAIRER(*bridge*)

```

1  start  $\leftarrow$  START(bridge)
2  end  $\leftarrow$  END(bridge)
3  island  $\leftarrow$  NEXT-ISLAND(start)
4  while island  $\neq$  end
5      do if  $\neg$ HAS-BRIDGE(island)
6          then if START-OF-BRIDGE(island)
7              then MEND-RIGHT(island, end)
8              else MEND-LEFT(start, island)
9          bridge  $\leftarrow$  BRIDGE(island)
10         BRIDGE-REPAIRER(bridge)
11         island  $\leftarrow$  NEXT-ISLAND(END(bridge))

```

Fig. 5. The BRIDGE-REPAIRER algorithm constructs artificial islands to match unmatched islands. The MEND-RIGHT algorithm is described further in Figure 6. The MEND-LEFT algorithm is symmetric to the MEND-RIGHT algorithm.

encountered. In this case there is a reef of the right type and value, before the enclosing island of type B, which points out a construction site.

The result of the BRIDGE-REPAIRER algorithm is shown in Figure 7. An artificial island of type B has been inserted to form a bridge with the previously unmatched island of type A.

```

MEND-RIGHT(broken, end)
1  node  $\leftarrow$  NEXT(broken)
2  while node  $\neq$  end
3      do if HAS-BRIDGE(node)
4          then node  $\leftarrow$  NEXT(BRIDGE-END(node))
5          else if POSSIBLE-CONSTRUCTION-SITE(broken, node)
6              then CONSTRUCT-ISLAND-AND-BRIDGE(broken, node)
7              return
8          else node  $\leftarrow$  NEXT(node)
9  CONSTRUCT-ISLAND-AND-BRIDGE(broken, PREVIOUS(end))

```

Fig. 6. The MEND-RIGHT algorithm constructs an artificial bridge end in the interval starting at the unmatched bridge start (*broken*) and ending at the end of the enclosing bridge (*end*)

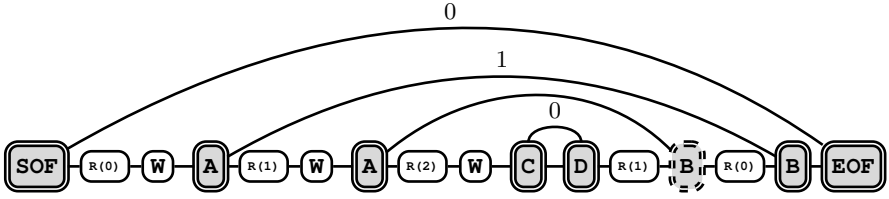


Fig. 7. The changes in the example bridge model after the BRIDGE-REPAIRER algorithm is done. The artificial island is marked with a dashed double edge.

4 Bridge Parser for Java

To construct a bridge parser for Java we need to create a bridge grammar which can provide information to each of the three parts of the bridge parser, illustrated in Figure 1. We will define this bridge grammar for Java in three steps which gradually will include more complex language-specific information. The performance impact of these different levels of language sensitivity is evaluated in Section 5.

For Java, and other languages with similar language constructs, we need to consider how to deal with comments and strings. These constructs might enclose text which would match as reefs or islands but which should be ignored. In our implementation we handle this separately in the lexer implementation.

4.1 Scopes

The first level of language sensitivity is to only consider tokens directly defining scopes. We therefore include braces and parentheses as islands since they have a direct impact on the nesting structure of Java code. Indentation is also included to enhance matching of islands in incomplete code. The complete bridge grammar looks like this:

Listing 1.4. Bridge Repairer Definitions

```

1  islands SOF, EOF, LBRACE, RBRACE, LPAREN, RPAREN
2  reefs INDENT(pos)
3
4  bridge from SOF to EOF
5
6  bridge from [a:INDENT LBRACE] to [b:INDENT RBRACE]
7      when a.pos = b.pos {
8          missing [RBRACE]
9              find [c:INDENT] where (c.pos <= a.pos) insert after
10         missing [LBRACE]
11             find [c:INDENT] where (c.pos <= a.pos) insert after
12     }
13
14 bridge from [a:INDENT LPAREN] to [b:INDENT RPAREN]
15     when a.pos = b.pos {
16         missing [RPAREN]
17             find [c:ISLAND] insert before
18             find [c:INDENT] where (c.pos <= a.pos) insert after
19         missing [LPAREN]
20             find [c:ISLAND] insert before
21             find [c:INDENT] where (b.pos <= c.pos) insert before
22     }

```

The `pos` attribute of the `INDENT` reef corresponds to the indentation level. Comparing two reefs of this type corresponds to comparing their `pos` attribute.

For the islands corresponding to right and left parentheses there are two `find` conditions. For cases like these the first occurrence that fulfills all its conditions decide which action to take.

4.2 Delimiters

To improve matching of parentheses we add additional reefs for delimiters. The following code snippet illustrates the benefit of defining commas as reefs during recovery:

```

void m(int a) {
    n(o( , a); // Recover to "n(o( ),a)" and not to "n(o(,a))"
}

```

We have a call to `o()` as the first argument in the call to `n()` in the method `m()`. The comma tells us that we are editing the first element in the list of arguments and that the call to `o()` should be closed right before the comma rather than after reading the a parameter. If we modify the code snippet and remove the other end parenthesis instead we end up with a different scenario:

```

void m(int a) {
  n(o(), a;    // Recover to "n(o(),a);" and not to "n(o(),a;)"
}

```

The analysis should ideally place a closing parenthesis somewhere after the comma but before the semicolon. The reason is that the comma separates elements within the parentheses and the call to `o()` is within the call to `n()`, while the semicolon separates elements in the block. To deal with these scenarios we define additional reefs to match delimiters and add additional **find** declarations to the missing parenthesis blocks.

Listing 1.5. Bridge Repairer Definitions

```

1  reefs .., COMMA, DOT, SEMICOLON
2
3  missing [RPAREN]
4    ..
5    find [c:COMMA] where (previous(c) != WATER) insert before
6    find [c:DOT] where (previous(c) != WATER) insert before
7    find [c:SEMICOLON] insert before
8
9  missing [LPAREN]
10   ..
11   find [c:COMMA] where (next(c) != WATER) insert after
12   find [c:DOT] where (next(c) != WATER) insert after
13   find [c:SEMICOLON] insert after

```

These definitions should be seen as extensions to the bridge grammar presented in the previous section. In the above **find** declaration for `RPAREN` we have added actions for when we encounter a `COMMA`, `DOT` or `SEMICOLON` while searching for a construction site.

4.3 Keywords

To further improve matching we can add keywords as reefs. This can be useful since keywords separate statements from each other. The following code snippet shows an example:

```

boolean m(boolean a) {
  if a == true    // Recover to "if (a == true)"
    return false; // and not to "(if a == true)"
  return true;
}

```

Ideally, the analysis should put the missing left parenthesis after the `if` keyword. If the keyword has been defined as a reef this is possible, otherwise not since then keywords will be considered to be water. To deal with scenarios such as these we add additional keywords and **find** definitions to the bridge grammar:

Listing 1.6. Bridge Repairer Definitions

```

1 reefs KEYWORD (if, for, while ..)
2
3 missing [RPAREN]
4     find [c:KEYWORD] insert before
5
6 missing [LPAREN]
7     find [c:KEYWORD] insert after

```

5 Evaluation

We have chosen to evaluate bridge parsing on common editing scenarios for Java using the specifications described in Section 4. Our bridge parsing implementation has been combined with a set of Java parsers, generated using state of the art parser generators based on LALR(1) and LL(variable lookahead) grammars. The parsers were generated with the following parser generators and settings:

- **Antlr** Generated using Antlr (v.3.0).
- **AntlrBT** Generated using a forth-coming version of Antlr (v.3.1 beta) with back-tracking turned on. This version of Antlr introduces new error recovery not yet available in the latest stable version.
- **Beaver** Generated with Beaver (v.0.9.6.1).
- **BeaverEP** Generated with Beaver (v.0.9.6.1), with error productions manually added to the grammar.
- **LPG** Generated using LPG (v.2.0.12). While this is the newest version of LPG it does not yet provide a complete automatic scope recovery as suggested by Charles[4]. To allow comparison with Charles approach the test cases were manually edited to correspond to the suggested recovery from the generated parser.

In lack of an existing appropriate benchmark suite we have created a test suite for incomplete and erroneous Java programs to use during evaluation. The test suite consists of several series of tests, each series with one *correct program* and one or more *broken programs*. The correct program in each series corresponds to the intention of the programmer, while the broken programs illustrate variations of the correct program where one or more tokens are missing. Broken programs in this setting illustrate how programs may evolve during an editing session. For each test series and parser we build a tree representing the correct program and try to do the same for each broken program. For the Antlr, AntlrBT and LPG we build parse trees, while for the Beaver and BeaverEP we build ASTs.

As a metric of how close a tree constructed from one of the broken programs is to the tree constructed for the correct program we use tree alignment distance, as described in [11]. To calculate tree alignment distance, a cost function is required which provides a cost for insertion, deletion and renaming of nodes. We use a simple cost function where all these operations have the cost one. As a complementary classification of success we use the categorization of recovery as excellent, good, or poor by Pennello and DeRemer [16].

5.1 Benchmark Examples

The test suite consists of 10 correct test cases which have been modified in various ways to illustrate possible editing scenarios. The test suite provides a total of 41 tests. Full documentation of the test suite can be found at [1]. We have focused on three editing scenarios:

Incomplete code. Normally, programs are written in a top-down, left-right fashion. In this scenario we put test cases with incomplete code, to illustrate how code evolves while being edited in an IDE. An example where a user is adding a method `m()` to a class `C` which already contains two fields `x` and `z` may look like this:

```
class C {  
    int x;  
    void m() {  
        int x;  
        if (true) {  
            int y;  
        }  
    }  
    int z;  
}
```

Missing start. This scenario highlights situations which might occur when the normal course of writing a program top-down, left-right is interrupted. A typical example is that the user goes back to fix a bug or to change an implementation in an existing program. Consider the example below where the programmer has started changing a while loop which causes a missing opening brace:

```
class C {  
    void m() {  
        // while (true) {  
            int a;  
        }  
    }  
}
```

Tricky indentation. Since bridge parsing relies on indentation it is reasonable to assume that tricky indentation is its Achilles heel. We therefore included a set of test cases with unintuitive indentation to evaluate its performance on such programs. An example of nested blocks where indentation is not increasing is shown below:

```
class C {  
    void m() {  
    }  
}
```


Another scenario leading to a similar situation is when a programmer pastes a chunk of code with different indentation in the middle of her program, as illustrated by this example:

```
class C {
    void n() { .. }
void m() {
}
}
```

5.2 Results

The results, after running through the test suite with our parser suite, are shown in Table 8, 9 and 10. The first table shows the results for tests with incomplete code, the second table shows results for tests with missing starts and the last table shows results for tests with tricky indentation.

Each table has five sets of four columns, one set for each parser generator. Each set contains a column with the tree editing distance without bridge parsing, with bridge parsing using the scopes version, with bridge parsing using the delimiters version and a final column summarizing the result for each test case. The results for each parser and test case are summarized with a letter indicating excellent (**E**), improved (**I**), status quo (**S**) or worse (**W**). A tree alignment distance of 0 indicates a full recovery and an excellent result while a missing value indicates total failure which is when no AST or parse tree could be constructed.

The leftmost column for each parser in the tables shows that the test suite presents many challenges to all parsers which manifest themselves in less than excellent recoveries, except for the test cases in Table 10 where only indentation is changed to trick the bridge parser. Without bridge parsing, LPG is much better than the other parser generators, most likely due to the built-in support for scope recovery.

The second column for each parser shows that all parser generators benefit vastly from bridge parsing in most cases, of 41 cases 19 improve from good or poor recovery to excellent. LPG still has the edge over the other generators with superior error recovery. We notice that using indentation can indeed improve scope recovery since the LPG results are improved in many cases.

The third column in each column set shows that there is almost no change when we use the delimiters version of the bridge parser instead of the scopes version. Generally, nothing changes or there are small improvement.

There are some problems with bridge parsing, as shown in Table 10, when there are inconsistencies in the layout. This problem could be alleviated by IDE support to automatically correct indentation during editing and pasting. Because of the current problems with some layout scenarios the bridge parser is only run when the parser fails to construct an AST and there is no other way to acquire an AST.

We have run tests with keyword sensitive bridge parsing as well, but saw no improvement using our test suite. There are certainly cases where this could yield an improvement but we could not easily come up with a convincing realistic editing scenario to include in the test suite.

Test	Antlr			AntlrBT			Beaver			BeaverEP			LPG							
A2	-	0	0	E	65	0	0	E	63	0	0	E	63	0	0	E	4	0	0	E
A3	75	0	0	E	1	0	0	E	63	0	0	E	63	0	0	E	0	0	0	E
A4	-	0	0	E	65	0	0	E	63	0	0	E	63	0	0	E	2	0	0	E
B1	75	0	0	E	28	0	0	E	73	0	0	E	30	0	0	E	0	0	0	E
B2	-	0	0	E	11	0	0	E	73	0	0	E	73	0	0	E	0	0	0	E
B3	29	2	0	E	29	1	0	E	33	-	0	E	33	-	0	E	8	2	1	E
B4	1	0	0	E	1	0	0	E	-	0	0	E	-	0	0	E	0	1	1	E
B5	75	0	0	E	1	0	0	E	73	0	0	E	73	0	0	E	8	0	0	E
C1	-	7	7	I	249	7	7	I	207	5	5	I	207	5	5	I	9	5	5	I
C2	249	0	0	E	29	0	0	E	207	0	0	E	123	0	0	E	0	0	0	E
C3	-	0	0	E	33	0	0	E	207	0	0	E	207	0	0	E	19	0	0	E
D1	168	0	0	E	114	0	0	E	124	0	0	E	81	0	0	E	12	0	0	E
D2	168	0	0	E	37	0	0	E	124	0	0	E	124	0	0	E	16	0	0	E
D3	168	0	0	E	65	0	0	E	124	0	0	E	105	0	0	E	2	0	0	E
D4	168	0	0	E	15	0	0	E	124	0	0	E	124	0	0	E	10	0	0	E
E1	31	-	-	W	28	18	17	I	109	109	109	S	47	47	109	W	18	12	10	I
E2	-	-	-	S	38	18	17	I	-	109	109	I	-	109	37	I	24	10	10	I
E3	125	0	0	E	16	0	0	E	109	0	0	E	109	0	0	E	11	0	0	E
F1	151	0	0	E	67	0	0	E	106	0	0	E	54	0	0	E	25	0	0	E
F2	151	0	0	E	44	0	0	E	106	0	0	E	54	0	0	E	9	0	0	E
F3	151	0	0	E	48	0	0	E	106	0	0	E	-	0	0	E	9	0	0	E
G1	1	0	0	E	1	0	0	E	-	0	0	E	-	0	0	E	0	0	0	E
G2	-	1	1	I	13	1	1	I	154	0	0	E	114	0	0	E	11	0	0	E
G3	-	-	-	S	36	34	34	I	154	154	154	S	154	154	154	S	2	2	2	S
H1	116	2	0	E	116	1	0	E	96	-	0	E	96	-	0	E	13	2	1	I
H2	-	2	0	E	97	1	0	E	73	11	0	E	73	11	0	E	16	2	0	E
H3	-	-	-	S	57	4	4	I	-	117	117	I	-	50	50	I	4	1	1	I
H4	-	-	-	S	8	7	7	I	117	15	15	I	117	15	15	I	13	5	5	I
I1	1	1	1	S	2	1	1	I	19	19	19	S	19	19	19	S	0	0	0	S
I2	2	1	1	I	2	1	1	I	21	21	21	S	21	21	21	S	15	15	15	S
I5	0	5	5	W	0	3	3	W	15	105	105	W	15	15	15	W	0	1	1	W
I4	0	0	0	E	0	1	1	W	15	-	-	W	15	-	-	W	0	3	3	W
I6	1	1	1	S	2	1	1	W	23	23	23	S	23	23	23	S	0	0	0	S

Fig. 8. Results for test cases with incomplete code

Test	Antlr				AntlrBT				Beaver				BeaverEP				LPG			
A5	9	0	0	E	9	0	0	E	63	0	0	E	63	0	0	E	5	0	0	E
C4	248	4	2	I	193	193	2	I	207	207	34	I	153	153	34	I	4	4	1	I

Fig. 9. Results for test cases with missing starts

Test	Antlr				AntlrBT				Beaver				BeaverEP				LPG			
A6	0	3	3	W	0	3	3	W	0	2	2	W	0	2	2	W	0	1	1	W
B6	0	10	10	W	0	64	64	W	0	73	73	W	0	23	23	W	0	4	4	W
I3	0	-	0	E	0	14	0	E	0	-	105	W	0	-	23	W	0	4	4	W
J1	0	17	17	W	0	59	59	W	0	49	49	W	0	49	49	W	0	15	15	W
J2	0	0	0	E	0	0	0	E	0	13	13	W	0	13	13	W	0	3	3	W

Fig. 10. Results for test cases with tricky indentation

6 Conclusions

We have presented bridge parsing as a technique to recover from syntactic errors in incomplete programs with the aim to produce an AST suitable for static semantic analysis. This enables tool developers to use existing parser generators when implementing IDEs rather than writing parsers by hand. The approach has proven successful when combined with several parser generators in our IDE generator for JastAdd based compilers in Eclipse.

The approach is highly general and can be used in combination with many different parsing technologies. We have validated this claim by showing how it improves error recovery for three different parser generators in common interactive editing scenarios.

One of the main goals of this work is to lower the burden on language developers who want to provide IDE support for their language. It is pleasant to notice that the language models for bridge parsing are very light-weight, yet yield good recovery on complex languages as exemplified by Java in this paper. We believe that it would be easy to adjust the bridge parser presented in this paper to support other languages as well.

As future work we would like to investigate the possibility of integrating history based information into the bridge model, work on improving the handling of incorrect layout and investigate how to derive bridge grammars from existing base-line grammars [12]. An other area we would like to look into is to improve the test suite by observing editing patterns passively from existing code and actively during development.

References

1. Bridge Parsing Test Suite (2008),
http://www.cs.lth.se/home/Emma.Nilsson_Nyman
2. Avgustinov, P., Ekman, T., Tibble, J.: Modularity First: A Case for Mixing AOP and Attribute Grammars. In: AOSD. ACM Press, New York (2008)
3. Burke, M.G., Fisher, G.A.: A practical method for LR and LL syntactic error diagnosis and recovery. ACM Trans. Program. Lang. Syst. 9(2), 164–197 (1987)
4. Charles, P.: A practical method for constructing efficient LALR(K) parsers with automatic error recovery. PhD thesis, New York, NY, USA (1991)
5. Degano, P., Priami, C.: Comparison of syntactic error handling in LR parsers, New York, NY, USA, vol. 25, pp. 657–679. John Wiley & Sons, Inc., Chichester (1995)
6. Earley, J.: An efficient context-free parsing algorithm. Commun. ACM 13(2), 94–102 (1970)

7. Ekman, T., Hedin, G.: The JastAdd Extensible Java Compiler. In: Gabriel, R.P. (ed.) 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007). ACM Press, New York (2007)
8. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation, vol. 39, pp. 111–122. ACM, New York (2004)
9. Graham, S.L., Haley, C.B., Joy, W.N.: Practical LR error recovery. In: SIGPLAN 1979: Proceedings of the 1979 SIGPLAN symposium on Compiler construction, pp. 168–175. ACM, New York (1979)
10. Huang, S.S., Hormati, A., Bacon, D.F., Rabbah, R.M.: Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 76–103. Springer, Heidelberg (2008)
11. Jiang, T., Wang, L., Zhang, K.: Alignment of trees - an alternative to tree edit. In: Theoretical Computer Science, vol. 143, pp. 137–148. Elsevier Science B.V., Amsterdam (1995)
12. Klusener, S., Lämmel, R.: Deriving tolerant grammars from a base-line grammar. In: ICSM 2003: Proceedings of the International Conference on Software Maintenance, Washington, DC, USA, p. 179. IEEE Computer Society, Los Alamitos (2003)
13. Moonen, L.: Generating robust parsers using island grammars. In: Proceedings. Eighth Working Conference on Reverse Engineering, pp. 13–22. IEEE Computer Society Press, Los Alamitos (2001)
14. Moonen, L.: Lightweight impact analysis using island grammars. In: Proceedings of the 10th IEEE International Workshop of Program Comprehension, pp. 219–228. IEEE Computer Society, Los Alamitos (2002)
15. Nilsson-Nyman, E., Ekman, T., Hedin, G., Magnusson, E.: Declarative Intraprocedural Flow Analysis of Java Source Code. In: 8th Workshop on Language Description, Tools and Applications. Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier Science, Amsterdam (2008)
16. Pennello, T.J., DeRemer, F.: A forward move algorithm for LR error recovery. In: POPL 1978: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 241–254. ACM, New York (1978)
17. Schäfer, M., Ekman, T., de Moor, O.: Sound and Extensible Renaming for Java. In: Kiczales, G. (ed.) 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008). ACM Press, New York (2008)
18. Tomita, M.: An efficient context-free parsing algorithm for natural languages and its applications. PhD thesis, Pittsburgh, PA, USA (1985)
19. van Deursen, A., Kuipers, T.: Building documentation generators. In: IEEE International Conference on Software Maintenance, pp. 40–49 (August 1999)
20. Wagner, T.A.: Practical Algorithms for Incremental Software Development Environments. PhD thesis, Berkeley, CA, USA (1998)
21. Wagner, T.A., Graham, S.L.: Efficient and flexible incremental parsing. *ACM Trans. Program. Lang. Syst.* 20(5), 980–1013 (1998)

Generating Rewritable Abstract Syntax Trees

A Foundation for the Rapid Development of Source Code Transformation Tools

Jeffrey L. Overbey and Ralph E. Johnson

University of Illinois at Urbana-Champaign, Urbana IL 61801, USA
{overbey2, johnson}@cs.uiuc.edu

Abstract. Building a production-quality refactoring engine or similar source code transformation tool traditionally requires a large amount of hand-written, language-specific support code. We describe a system which reduces this overhead by allowing both a parser and a *fully rewritable* AST to be generated automatically from an *annotated* grammar, requiring little or no additional hand-written code. The rewritable AST is ideal for implementing program transformations that preserve the formatting of the original sources, including spacing and comments, and the system can be augmented to allow transformation of C-preprocessed sources even when the target language is not C or C++. Moreover, the AST design is fully customizable, allowing it to resemble a hand-coded tree. The amount of required annotation is typically quite small, and the annotated grammar is often an order of magnitude smaller than the generated code.

1 Introduction

To many programmers, the *Refactor* > *Extract Method* menu item is “just another feature” of an IDE, one with the same visual precedence as *Edit* > *Copy* or *File* > *Print*. However, to the IDE developer, these features are not at all comparable: Building a refactoring tool is a substantial development effort. Moreover, it requires a substantial amount of infrastructure, most of which is usually written by hand.

The central data structure in a refactoring engine is usually a *rewritable AST*, that is, an abstract syntax tree whose nodes can be added to, rearranged, removed, and replaced *to perform textual transformations on the underlying source code*. Nearly all of a refactoring engine’s components depend on the AST, so it must be implemented before any significant program transformations or analyses. And, unfortunately, it is a component that is often very time-consuming to code by hand.

This paper describes a system that generates an abstract syntax tree from a grammar. The grammar can be *annotated* to customize the AST design, and then it is used to generate AST node classes as well as a parser which constructs ASTs comprised of these nodes. The generated ASTs are *rewritable* in the sense that refactorings and other source code transformations can be coded by restructuring the AST, and the revised source code can be emitted, preserving all of the user’s formatting, including spacing and comments; there is no need to develop a prettyprinter. The system can even be extended to handle preprocessed code. Moreover, while the generated AST nodes

are usually comparable to hand-coded AST nodes, they can be customized, replaced, or intermixed with hand-coded AST nodes. This AST generation system has been implemented in a tool called Ludwig and has been used to generate rewritable ASTs for several projects, most notably Photran, an open source refactoring tool for Fortran.

The remainder of this paper is organized around our three major contributions. §2 describes how to annotate a parsing grammar to produce a satisfactory AST. Although many AST generators already exist, our system of annotations is new and is fundamentally different from (and arguably more concise than) all of the existing AST specification languages. §3 describes how to augment an AST to allow both rewriting and faithful reproduction of the original source code. Although it is fairly straightforward, this process (“concretization”) also appears to be new. §4 discusses modifications necessary to support preprocessed source text. This is the first work, to our knowledge, that addresses supporting C preprocessor directives in arbitrary languages; here, our major contribution is in how we handle conditional compilation. §5 gives empirical results on the ASTs we generate; our work is compared and contrasted with related work in §6, and the paper concludes in §7.

2 Abstract Syntax Annotations

The grammar supplied to a parser generator defines the *concrete* syntax of the language, which is almost always different from an ideal *abstract* syntax. For example, consider the following grammar for a language where a program is simply a list of statements, and a statement is either an if-statement, an unless-statement, or a print-statement.

$\langle \text{program} \rangle$	$::= \langle \text{program} \rangle \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle$	(1)
$\langle \text{stmt} \rangle$	$::= \langle \text{if-stmt} \rangle \mid \langle \text{unless-stmt} \rangle \mid \langle \text{print-stmt} \rangle$	(2)
$\langle \text{if-stmt} \rangle$	$::= \text{IF } \langle \text{expr} \rangle \text{ THEN } \langle \text{stmt} \rangle \text{ ENDIF}$	(3)
	$\mid \text{IF } \langle \text{expr} \rangle \text{ THEN } \langle \text{stmt} \rangle \text{ ELSE } \langle \text{stmt} \rangle \text{ ENDIF}$	(4)
$\langle \text{unless-stmt} \rangle$	$::= \text{UNLESS } \langle \text{expr} \rangle \langle \text{stmt} \rangle$	(5)
$\langle \text{print-stmt} \rangle$	$::= \text{PRINT } \langle \text{expr} \rangle$	(6)
	$\mid \text{PRINT } \langle \text{expr} \rangle \text{ TO STDERR}$	(7)
$\langle \text{expr} \rangle$	$::= \text{TRUE} \mid \text{FALSE} \mid \text{LITERAL-STRING}$	(8)

There are several ways to generate an AST directly from the grammar. Clearly, such an AST will not have an ideal design, but it is useful as a starting point; in this section, we will define six annotations which allow us to refine the AST’s design. The AST nodes we generate will be classes comprised of public fields, although adapting our technique to generate properly-encapsulated classes or even non-object-oriented ASTs (e.g., using structs and unions in C or algebraic data types in ML) is straightforward.

One obvious method for generating an AST directly from the grammar is to

- generate one AST node class for each nonterminal, where
- this class contains one field for each symbol on the right-hand side of one of that nonterminal’s productions.

For example, the AST nodes corresponding to $\langle \text{print-stmt} \rangle$ and $\langle \text{stmt} \rangle$ would be the following, where *Token* is the name of a class representing tokens returned by a lexical analyzer.

```

class PrintStmtNode {
    public Token print ;
    public ExprNode expr;
    public Token to ;
    public Token stderr ;
}

class StmtNode {
    public IfStmtNode ifStmt ;
    public UnlessStmtNode unlessStmt;
    public PrintStmtNode printStmt ;
}

```

When these classes are instantiated in an AST, irrelevant fields will be set to null.

2.1 Annotation 1: Omission

This method for generating ASTs has several things wrong with it. One of the most obvious is that keywords like IF and PRINT are almost never included in an AST. We will indicate that these tokens can be *omitted* or *elided* by ~~striking out~~ their symbols in the grammar.¹ For example,

$$\begin{aligned}
 \langle \textit{print-stmt} \rangle &::= \textit{PRINT} \langle \textit{expr} \rangle \\
 &| \textit{PRINT} \langle \textit{expr} \rangle \textit{TO} \textit{STDERR}
 \end{aligned}$$

would generate a *PrintStmtNode* with only two fields: *expr* and *stderr*.

Generalizing this slightly, we will establish the following rules.

- If a symbol on the right-hand side of a production is annotated for omission, no field is generated for that symbol.
- If a nonterminal on the left-hand side of a production is annotated for omission, no AST node class is generated for that nonterminal.²

2.2 Annotation 2: Labeling

Determining the names of AST classes and their fields from the names of nonterminal and terminal symbols in the grammar is often sufficient, but sometimes these names need to be customized. This can be done by *explicitly labeling* symbols in the grammar and interpreting these labels as follows.

- Labeling the nonterminal on the left-hand side of a production determines the name of the AST node class to generate.
- Labeling a nonterminal or terminal symbol on the right-hand side of a production determines the name of the field to which that symbol corresponds.

The idea of labeling symbols is simple, yet it is extremely powerful, having several uses and implications.

¹ In our implementation, we represent the same in ASCII by prefixing the symbol with a hyphen and a colon, as in `-:print`.

² The ability to omit entire AST node classes is useful when only a partial AST is desired. For example, the Eclipse JDT [10] and CDT [9] both contain a “lightweight” AST which describes high-level organizational structures (classes, methods, etc.) but not statements or expressions.

Labeling to Distinguish Fields. Labeling is the only annotation that is strictly required, at least in certain cases. *IfStmtNode* is one example. The problem is production (4):

$$\langle \text{if-stmt} \rangle ::= \text{IF } \langle \text{expr} \rangle \text{ THEN } \langle \text{stmt} \rangle \text{ ELSE } \langle \text{stmt} \rangle \text{ ENDIF}$$

Since the symbol $\langle stmt \rangle$ appears twice in the same production, we need *two* fields so that the node can have separate fields for the then-statement and the else-statement. We will accomplish this by adding distinctive labels to these symbols in the grammar, rewriting productions (3) and (4) to generate distinct fields for the two occurrences of $\langle stmt \rangle$.

$$\begin{array}{c} \text{thenStm} \\ \langle \text{if-stm} \rangle ::= \text{IF } \langle \text{expr} \rangle \text{ THEN } \langle \text{stm} \rangle \text{ ENDIF} \\ \text{thenStm} \quad \text{elseStm} \\ \text{IF } \langle \text{expr} \rangle \text{ THEN } \langle \text{stm} \rangle \text{ ELSE } \langle \text{stm} \rangle \text{ ENDIF} \end{array}$$

This will generate the following node instead.

```
class IfStmtNode {
    public ExprNode expr;
    public StmtNode thenStmt;
    public StmtNode elseStmt;
}
```

Labeling to Merge Fields. Just as we can give symbols distinct labels to create distinct fields, we can also give several several symbols the *same* label to assign them to the *same* field ... as long as they occur in different productions. One example of this appears above: Since both occurrences of `<stmt>` were given the label *thenStmt*, the `thenStmt` field will be populated regardless of whether production (3) or (4) was matched.

Labeling to Rename Fields. Labeling can also be used to avoid illegal or undesirable field names. For example, were the `if` token not omitted, it would generate a field named `if`, which is illegal in Java, so it could be labeled *ifToken* instead. Similarly, `<expr>` might be labeled *guardingExpression* to make its corresponding field name more descriptive.

Labeling to Distinguish, Rename, and Merge Node Classes. Just as labeling the symbols on the right-hand sides of productions allows us to distinguish, rename, and merge fields, labeling the nonterminals on the left-hand sides of productions allows us to distinguish, rename, and merge AST node classes.

The text of the label assigned to a left-hand nonterminal determines the name of the AST node class generated for that nonterminal. By assigning a distinct label to each left-hand nonterminal, we ensure that a different AST node class will be generated for each nonterminal. By assigning the same label to several left-hand nonterminals, they can all correspond to the same AST node class.

But when is it useful to have just one node class correspond to several nonterminals?

Sometimes a grammar uses several nonterminals to refer to the same logical entity. This is probably most common in expression grammars, as we will see later, but we can illustrate it with our sample grammar. Note that our sample programming language

contains two conditional constructs: an if-statement and an unless-statement. Suppose we want to represent both of these with a single node, *ConditionalStmtNode*. We will label the *left-hand* nonterminals with this name

$$\begin{array}{lcl}
 \text{ConditionalStmtNode} & & \text{thenStmt} \\
 \underbrace{\langle \text{if-stmt} \rangle} & ::= & \text{IF } \langle \text{expr} \rangle \text{ THEN } \langle \text{stmt} \rangle \text{ ENDIF} \\
 & & \text{thenStmt} \quad \text{elseStmt} \\
 & | & \text{IF } \langle \text{expr} \rangle \text{ THEN } \langle \text{stmt} \rangle \text{ ELSE } \langle \text{stmt} \rangle \text{ ENDIF} \\
 \text{ConditionalStmtNode} & & \text{elseStmt} \\
 \underbrace{\langle \text{unless-stmt} \rangle} & ::= & \text{UNLESS } \langle \text{expr} \rangle \langle \text{stmt} \rangle
 \end{array}$$

in order to generate a *ConditionalStmtNode* class with the same three fields as before (cf. page 117). When an if-statement is matched, the *expr* and *thenStmt* fields will be set, and the *elseStmt* field may or may not be null. When an unless-statement is matched, *expr* and *elseStmt* will be set, but *thenStmt* will always be null.³

2.3 Annotation 3: Boolean Access

Consider the print-statement, defined in productions (6) and (7). Suppose that a print-stmt can write either to standard output or standard error, depending on whether or not the **to stderr** clause is present. Although we can omit the **PRINT** and **TO** tokens from the *PrintStmtNode* class, we cannot also omit the **STDERR** token without losing the semantic distinction between the two variants of the print-statement. The token **STDERR** is not important *per se*; what matters is whether or not it was present at all.

We will annotate a symbol with “(bool)” to indicate that its field should be of type boolean rather than of type *Token*. When the corresponding token is present, the field’s value will be set to true; otherwise, it will be set to false. Accordingly,⁴

$$\begin{array}{lcl}
 \langle \text{print-stmt} \rangle & ::= & \text{PRINT } \langle \text{expr} \rangle \\
 & | & \text{PRINT } \langle \text{expr} \rangle \text{ TO } \text{STDERR}
 \end{array}$$

(bool)

will generate the node class

```

class PrintStmtNode {
    public ExprNode expr;
    public boolean stderr ;
}

```

whose *stderr* field can be tested to determine whether the print-statement is intended to write to standard output or standard error.

2.4 Annotation 4: List Formation

Recursive productions are idiomatically used to specify lists. In our example grammar, the productions on line (1) indicate that a program is a list of one or more statements. In

³ This is based on the intuition that **unless** *E S* is equivalent to **if** *E* **then no-op else** *S*.

⁴ Again, our implementation uses a straightforward ASCII equivalent: (bool):stderr. Also, one should note that some obvious extensions of this annotation are possible, such as ones to assign an integer or an *enum* value.

an AST, it is usually preferable to replace these recursive structures with an array, list, or whatever iterable construct is most common in the implementation language.

We will annotate left-hand nonterminals with “(list)” to indicate that the productions for that nonterminal describe a list.

$$\overbrace{\langle \text{program} \rangle}^{(\text{list})} ::= \langle \text{program} \rangle \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle$$

Now, there is no need to generate a *ProgramNode* class: In its place, we can simply use a *List<StmtNode>*.

2.5 Annotation 5: Inlining

To illustrate our next annotation, suppose the productions defining an if-statement had been written as follows. Note that the syntax of the if-statement has not changed: The grammar is just slightly different.

$$\begin{aligned} \langle \text{if-stmt} \rangle &::= \langle \text{if-then-part} \rangle \langle \text{endif-part} \rangle \\ &\quad \mid \langle \text{if-then-part} \rangle \overbrace{\langle \text{else-part} \rangle}^{\text{thenStmt}} \langle \text{endif-part} \rangle \\ \langle \text{if-then-part} \rangle &::= \text{IF } \langle \text{expr} \rangle \text{ THEN } \overbrace{\langle \text{stmt} \rangle}^{\text{thenStmt}} \\ \langle \text{else-part} \rangle &::= \text{ELSE } \overbrace{\langle \text{stmt} \rangle}^{\text{elseStmt}} \\ \langle \text{endif-part} \rangle &::= \text{ENDIF} \end{aligned}$$

Compare these to productions (3) and (4) in the original sample grammar. In this version, IF $\langle \text{expr} \rangle$ THEN $\langle \text{stmt} \rangle$ has been “factored out” into its own nonterminal, $\langle \text{if-then-part} \rangle$. This is commonly done to minimize duplication in the grammar. However, left alone, it adds unnecessary nodes to an AST. In this case, there will be *three* AST nodes, all devoted to defining the structure of an if-statement: *IfStmtNode*, *IfThenPartNode*, and *ElsePartNode*.⁵

To create the same nodes as before, we would *like* to do away with *IfThenPartNode* and *ElsePartNode* and instead have their fields—*expr*, *thenStmt*, and *elseStmt*—placed directly into the *IfStmtNode* class. In other words, we would like to *inline* these nodes: In the *IfStmtNode* class, rather than declaring an *IfThenPartNode* field, we will simply insert all of the fields that would be in an *IfThenPartNode* instead. We will denote this with an “(inline)” annotation

$$\begin{aligned} \langle \text{if-stmt} \rangle &::= \overbrace{\langle \text{if-then-part} \rangle}^{(\text{inline})} \langle \text{endif-part} \rangle \\ &\quad \mid \overbrace{\langle \text{if-then-part} \rangle}^{(\text{inline})} \overbrace{\langle \text{else-part} \rangle}^{(\text{inline})} \langle \text{endif-part} \rangle \\ \langle \text{if-then-part} \rangle &::= \text{IF } \langle \text{expr} \rangle \text{ THEN } \overbrace{\langle \text{stmt} \rangle}^{\text{thenStmt}} \\ \langle \text{else-part} \rangle &::= \text{ELSE } \overbrace{\langle \text{stmt} \rangle}^{\text{elseStmt}} \\ \langle \text{endif-part} \rangle &::= \text{ENDIF} \end{aligned}$$

which gives us the desired AST node. Notice that we have now omitted $\langle \text{if-then-part} \rangle$ and $\langle \text{else-part} \rangle$: Since their contents are always inlined, there is no reason to generate these node classes.

⁵ Notice that we have omitted $\langle \text{endif-part} \rangle$, since its AST node is unnecessary.

```

class IfStmtNode {
    public ExprNode expr;           // Inlined from IfThenPartNode
    public StmtNode thenStmt;       // Inlined from IfThenPartNode
    public StmtNode elseStmt;       // Inlined from ElsePartNode
}

```

2.6 Annotation 6: Superclass Formation

Idiomatic Form. The productions in line (2) illustrate another common idiom in BNF grammars:

$$\langle stmt \rangle ::= \langle if-stmt \rangle \mid \langle unless-stmt \rangle \mid \langle print-stmt \rangle$$

states that an if-statement is a statement, an unless-statement is a statement, and a print-statement is a statement. In object-oriented languages, this *is-a* relationship is generally modeled using inheritance. Instead of generating a *Stmt* node with fields for the various types of statements, we can instead make *Stmt* an abstract class (or interface, in Java or C[#]) which is *subclassed* by *IfStmt*, *UnlessStmt*, and *PrintStmt*. We will indicate this preference by a “(superclass)” annotation on the left-hand nonterminal.

$$\overset{\text{(superclass)}}{\langle stmt \rangle} ::= \langle if-stmt \rangle \mid \langle unless-stmt \rangle \mid \langle print-stmt \rangle$$

This generates the following.⁶

```

interface StmtNode { /*empty*/ }
class IfStmtNode implements StmtNode { ... }
class UnlessStmtNode implements StmtNode { ... }
class PrintStmtNode implements StmtNode { ... }

```

Non-idiomatic Form. As in the preceding example, the (superclass) annotation is generally applied to a nonterminal whose productions are all of the form $A ::= B$, for nonterminals A and B . But it is also possible to apply this annotation when the productions do not have this form. To do this, we must explicitly label each *production* with a node class name. For example,

$$\overset{\text{(superclass)}}{\langle if-stmt \rangle} ::=$$

$$\{ \{ \langle expr \rangle \text{ THEN } \overset{\text{thenStmt}}{\langle stmt \rangle} \text{ ENDIF } \} \text{ IfThenNode} \mid \{ \langle expr \rangle \text{ THEN } \overset{\text{thenStmt}}{\langle stmt \rangle} \text{ ELSE } \overset{\text{elseStmt}}{\langle stmt \rangle} \text{ ENDIF } \} \text{ IfThenElseNode} \}$$

allows us to have two *different* nodes for an if-statement, one for the if-then form and another for the if-then-else form.

⁶ In our implementation, the interface/abstract superclass contains no fields or methods, as shown here. Later in this paper, we describe several ways to customize generated nodes. This empty interface is often an excellent candidate for customization, since certain behaviors may be common among the various subclasses.

```

interface IfStmtNode { /*empty*/ }

class IfThenNode
implements IfStmtNode {
  public ExprNode expr;
  public StmtNode thenStmt;
}

class IfThenElseNode
implements IfStmtNode {
  public ExprNode expr;
  public StmtNode thenStmt;
  public StmtNode elseStmt;
}

```

It should be noted that the labeling principles discussed in §2.2 apply to production labels as well. For example, two productions (or a production and a nonterminal) can be given the same label to assign them to the same node class.

2.7 Customization

In our experience, these annotations—omission, labeling, Boolean access, list formation, inlining, and superclass formation—allow satisfactory AST nodes to be generated for most constructs in most languages. However, it is sometimes desirable to “tweak” some of the generated AST node classes or to mix them with non-generated nodes. Our implementation [24] provides several means to customize the generated AST; the reader is referred to its documentation for details. In our experience, the two most important customizations are the following.

- *The user must be able to add methods to the generated node classes.* For example, it may be desirable to add a `getType()` method to expression nodes or a `resolveBinding()` method to identifier nodes. Our implementation follows a best practice described by Vlassides [35, p. 85]: The system generates an AST node class, and the user places additional methods in a custom subclass (or superclass).
- *The user must be able to write custom AST nodes when necessary.* Sometimes, the “obvious” grammatical representation of a language construct does not satisfy the constraints of the parser generator; this can result in productions which deviate wildly from the conceptual structures of the constructs they are intended to represent. In these cases, the user must be able to hand-code an AST node for that construct. Being able to intermix hand-coded and generated nodes is critical, because it provides the user with the flexibility of hand-coding when it is needed while alleviating the tedium and cost of developing and maintaining an entirely hand-coded infrastructure.

2.8 An Example

We will conclude this section with a fairly complex example: Using annotations to construct an AST for an expression grammar.

$$\begin{array}{lcl}
 \begin{array}{c} \text{IExpression} \\ \text{(superclass)} \end{array} & \begin{array}{c} \text{lhs} \quad \text{isPlus} \\ \text{(boolean)} \quad \text{rhs} \end{array} & \\
 \underbrace{\langle \text{expr} \rangle} & ::= & \underbrace{\langle \text{expr} \rangle}_{\text{lhs}} \text{ PLUS } \underbrace{\langle \text{term} \rangle}_{\text{rhs}} \quad \} \text{ BinaryExpr} \quad (1) \\
 & | & \langle \text{term} \rangle \quad (2) \\
 \begin{array}{c} \text{IExpression} \\ \text{(superclass)} \end{array} & \begin{array}{c} \text{lhs} \quad \text{isTimes} \\ \text{(boolean)} \quad \text{rhs} \end{array} & \\
 \underbrace{\langle \text{term} \rangle} & ::= & \underbrace{\langle \text{term} \rangle}_{\text{lhs}} \text{ TIMES } \underbrace{\langle \text{factor} \rangle}_{\text{rhs}} \quad \} \text{ BinaryExpr} \quad (3) \\
 & | & \langle \text{factor} \rangle \quad (4)
 \end{array}$$

$$\begin{array}{lcl}
 \overbrace{\langle factor \rangle}^{IExpression \text{ (superclass)}} & ::= & \overbrace{\langle factor \rangle}^{lhs} \overbrace{EXP}^{isExp \text{ (boolean)}} \overbrace{\langle primary \rangle}^{rhs} \} BinaryExpr \\
 & | & \langle primary \rangle
 \end{array}
 \quad \begin{array}{l} (5) \\ (6) \end{array}$$

$$\overbrace{\langle primary \rangle}^{IExpression \text{ (superclass)}} ::= \langle constant \rangle \quad (7)$$

$$\langle primary \rangle ::= \text{LPAREN} \langle expr \rangle \text{RPAREN} \quad (8)$$

$$\langle constant \rangle ::= \text{INTEGER-CONSTANT} \quad (9)$$

$$\langle constant \rangle ::= \text{REAL-CONSTANT} \quad (10)$$

The preceding grammar is a stereotypical example of an unambiguous grammar for arithmetic expressions. From lowest to highest precedence, it incorporates addition (left-associative), multiplication (left-associative), exponentiation (right-associative), and nested expressions. Annotations are used as follows.

- *Superclass Formation.* Production (1) indicates that a superclass, *IExpression*, will be generated, and that expressions of the form $\langle expr \rangle \text{ PLUS } \langle term \rangle$ will be parsed into a node called *BinaryExpression* which implements *IExpression*. Production (2) indicates that the AST node class for $\langle term \rangle$ will also implement *IExpression*; however, since the AST node class for $\langle term \rangle$ is *IExpression*, this has no effect. Likewise, production (7) indicates that *Constant*, the AST node class for $\langle constant \rangle$, will implement *IExpression*. Production (8) indicates that the AST node for $\langle expr \rangle$ should implement *IExpression*, but, again, this is trivially true.
- *Labeling.* Labels are used to name the fields in *BinaryExpr*; they are used to assign the same AST node class, *IExpression*, to $\langle expr \rangle$, $\langle term \rangle$, $\langle factor \rangle$, and $\langle primary \rangle$; and they are used to assign productions (1), (3), and (5) to the same node class, *BinaryExpr*. The parentheses delineating a nested expression in (8) are omitted, so they will not be included in the AST.
- *Boolean Access.* The tokens PLUS, TIMES, and EXP are assigned Boolean fields in the *BinaryExpr* node class.⁷

Ultimately, this results in the following node classes.

```

interface IExpression { /* empty */ }

class BinaryExpr implements IExpression {
    public IExpression lhs;
    public IExpression rhs;
    public boolean isPlus;
    public boolean isTimes;
    public boolean isExp;
}

class Constant implements IExpression {
    public Token integerConstant;
    public Token realConstant;
}

```

⁷ This was primarily for illustrative purposes. We could have given productions (1), (3), and (5) different labels to have separate nodes types for addition, multiplication, and exponentiation expressions.

3 Rewritable Abstract Syntax Trees

We will next present how to generate *rewritable* ASTs, which allow source code to be modified simply by adding, changing, moving, and removing nodes in the AST.

Traditionally, source code is modified either by prettyprinting a modified AST or by computing textual edits from an AST. Prettyprinting is a popular choice for source-to-source compilers and academic/research source code transformation tools (including Stratego/XT [34,7], TXL [5], ASF+SDF [33], and many academic refactoring tools): Prettyprinting is straightforward to implement, and preserving comments and source formatting is usually a non-goal. On the other hand, commercial refactoring tools (including the Eclipse JDT [10] and CDT [9], NetBeans [2], and Apple Xcode [3]) generally use textual edits: AST nodes are mapped to offset/length regions in the source file, and the source text is changed by manipulating a text buffer directly (by specifying text to be added, removed, or replaced at particular offsets) or indirectly (by manipulating AST nodes and computing text buffer changes from the AST modifications). Prettyprinting is easier to implement but sacrifices output quality; textual edits produce “better” results at the expense of a significantly more complicated implementation.

Our solution fuses these two approaches: We will add “missing pieces”—dropped tokens, spaces, comments, etc.—back into the AST, but they will not be visible in its public interface. This will allow us, internally, to reproduce the original source text exactly using a simple traversal. Moreover, they will be tied to individual nodes in such a way that they move *with* the nodes when the AST is modified.

3.1 Whitetext Augmentation

If an AST contains every token in the original text, in the original order, the original source code can be reproduced almost exactly. The only pieces missing are what we refer to as *whitetext*: spaces, comments, line continuation characters, and similar lexical constructs. In order to reproduce the original text *exactly*—including whitetext—we propose the following.

- Rather than discarding whitetext, the lexical analyzer must “attach” all whitetext to exactly one token: either the token preceding it or the one following it.
- Most whitetext is attached to the *following* token.
- However, whitetext appearing at the end of a line is considered to be part of the *preceding* token, along with the carriage return/linefeed following it. Any additional whitetext beyond the carriage return/linefeed is attached to the *following* token.

The latter part of this heuristic ensures that any trailing comments on a line, as well as the carriage return/linefeed itself, are associated with the preceding token, while any subsequent indentation is associated with the token on the next line.

3.2 AST Concretization

Note that our heuristic uses tokens to *partition* the original source text: Every character in the original text is also present in a token, every character in a token is also present

in the original text, and there is no overlap between tokens. Moreover, the characters retain their original source text order.⁸

Since a whitetext-augmented token stream partitions the original source text, it can be used to reproduce exact source text. Thus, it should also be possible to reproduce source text from an AST ... as long as every token is present, in the original order. Reviewing our list of AST annotations, we can see that this is not necessarily the case:

1. Tokens may be omitted.
2. Node classes may be omitted.
3. Nodes may be replaced with Boolean values.
4. A node's children are assigned to named fields; however, since several productions may be assigned to a single type of node, there is not necessary a single order in which these children can be traversed to preserve the original order. (For example, consider the node generated for the productions $A ::= ab \mid ba.$)

We can overcome these problems with the following, respectively.

1. Rather than omitting tokens from node classes, store them in a *private* field, making them accessible for source text reproduction while remaining absent from the node's interface.
2. When an omitted node class is used (and not inlined), simply store a string containing the node's original text or, equivalently, a list/array of tokens.
3. Rather than storing a Boolean value, store the original node/token, and provide a Boolean accessor method for that field.
4. If there *is* a single order in which children can be traversed to preserve the original token order, there is no problem; if no such order exists, then the node must include a field indicating the appropriate traversal order.

One method to determine an order for printing a node's children is the following. Each production generating a particular node defines a partial order on some of that node's children. The union of these partial orders is a relation describing the ordering of all of the node's children. We may treat this relation as a directed graph and topologically sort it using an algorithm that also detects cycles [6, p. 546]. If no cycles are found, the sorted graph gives a correct (total) order for printing the node's fields; if cycles are found, no such order exists.

3.3 AST Rewriting

When the above changes are made, it is possible to reproduce the exact source code from which an AST was created simply by traversing the tree and outputting each token and its associated whitetext. However, such a tree is also ideal for *rewriting*. When a node is moved or removed within the tree, every token under that node—omitted or not—is moved with it, as are any comments and line endings associated with that

⁸ This assumes that there is at least one token in the original file. If the original source consists solely of whitetext—say, a C program that contains only a comment—this must be treated as a special case.

node. Suppose, for example, that an *IfStmtNode* is moved to a new location in the AST: When the modified AST is traversed to reproduce source code, the entire if-statement—including the `if` token, the line ending, and any comments—will appear at the new location within the source code.⁹

4 Representing Preprocessed Text

So far, we have ignored one mundane but highly nontrivial aspect of source code transformation: handling preprocessed code. Many tools—including `sed`, `m4`, and even Perl—can serve as preprocessors, making the topic impossible to treat in full generality. Thus, we will focus on one of the most commonly used tools: the C preprocessor. Its capabilities are stereotypical, so the conceptual model we develop is applicable, for example, when handling `INCLUDE` lines in Fortran or simple `sed` substitutions as well. We will assume that the reader is already familiar with the C preprocessor; a good introduction is provided in the GCC documentation [31], while the C and C++ language specifications [18,19] provide normative references and are the source of the C preprocessor terminology in this section.

We will follow the *pseudo-preprocessing* model of [14]: We will assume that a customized version of the preprocessor will feed the lexical analyzer. While a normal preprocessor simply executes preprocessing directives and outputs either a stream of text or a stream of tokens, our “pseudo-preprocessor” will keep track of what output originated from what directives, passing this information on to the lexer (and parser) so that it can be incorporated into the AST. Retaining information about preprocessor directives in the AST is essential both for analyzing preprocessed code and for reproducing the original (un-preprocessed) source code from the AST.

The C preprocessor has three aspects to consider: substitutions, control lines, and conditional compilation.

4.1 Substitutions

Trigraphs,¹⁰ backslash-newline sequences, `#include` directives, and macro expansions are all essentially textual substitutions, i.e., the logical text for some tokens may differ from the physical text in the unpreprocessed file. These can be handled using a technique essentially similar to Garrido’s [14], marking the affected tokens with the token’s original text (or original preprocessor directive) from the unpreprocessed file and printing this, rather than the token’s logical text, during source reproduction.

4.2 Control Lines

Control lines include macro definitions (`#define` and `#undef`), the line control directive (`#line`), `#error`, `#pragma`, and the null directive (`#` followed by a newline).¹¹ We will

⁹ One should note that the if-statement will retain its indentation from the previous location. If the level of indentation needs to be adjusted, this must be done manually by modifying the whitext of the tokens under the affected node.

¹⁰ A trigraph is a sequence of three characters that takes the place of another character. For example, `???` is equivalent to `{`.

¹¹ The `#include` directive is also a control line but is treated uniquely for our purposes.

treat these as whitetext, but how we do so depends on the application. When source reproduction is the only goal, they can be treated as strings and affixed to tokens like spaces and comments. When some analysis is required, it is useful to represent them with nodes in the AST. One must note that inserting a directive in the middle of a statement

```
int
#define MACRO
some_function() {}
```

is perfectly legal; directives do not necessarily appear at statement boundaries. This may be particularly true when languages other than C and C++ are preprocessed using the C preprocessor. So in general, no assumptions can be made about where directives will or will not need to be placed in an AST.

One means to include nodes for these directives in the AST is to make tokens' associated whitetext a *sequence* of strings and preprocessor directives. The preprocessor directives can be treated as children of tokens during a traversal; this makes it possible to include them, for example, when implementing the Visitor pattern [12].

4.3 Conditional Compilation and Multiple Configurations

Conditional compilation is by far the most challenging feature to handle in a source code transformation tool. Consider the following example.

```
if (
  #if defined(A) || defined(B)
  variable
  #else
  function() < 1 && variable
  #endif
  < 2) x = 3;
```

We cannot treat conditional directives as whitetext (as we did for control lines) because doing so results in the token sequence

```
if (variable function() < 1 && variable < 2) x = 3;
```

which will not even parse.

Conceptually, conditional inclusion allows for variation in the AST depending on the preprocessor's *configuration*, that is, the set of macro definitions under which it is operating: The structure of the AST under one configuration is not necessarily the same as for another. If we intend to reproduce the original source code from an AST, we must be able to construct a single AST which captures *all* of the AST variations that may occur under *all* feasible configurations.

Representation. Garrido's [14] technique for handling conditional compilation involves modifying the grammar to specify where conditionals may appear and then inserting a pass between lexical analysis and parsing which ensures that conditional directives appear only at these locations: If a conditional directive appears at an unexpected location, it is moved forward or backward as necessary, and tokens are copied into each branch of the conditional. This process is called *completing the conditional*.

Although reasonable, it is language-specific, heuristic, and becomes complicated in the presence of nested conditionals. It also requires modifying the grammar and adding AST nodes for conditional compilation directives.

Our solution for representing multiple-configuration sources, which is intended to be language-agnostic and grammar-independent, is based on [32, p. 5], which uses a similar structure to represent ambiguities in parse trees for natural language. Figures 1(a) and (b) show the individual ASTs that would be constructed for the preceding example under each preprocessor configuration. Notice that the smallest subtree that differs between the two is the expression under the if-statement node. Figure 1(c) shows a “multiple-configuration” AST which combines the previous ASTs by inserting an “ambiguous” expression node whose children are the individual expression nodes guarded by the various preprocessor configurations.

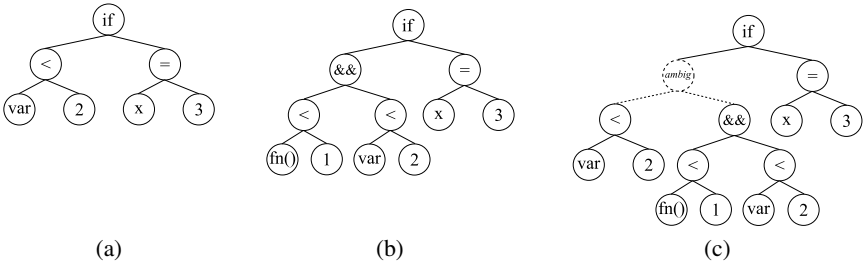


Fig. 1. (a) The AST for the configuration `defined(A) || defined(B)`. (b) The AST for the configuration `!defined(A) || defined(B)`. (c) The multiple-configuration AST. The dotted node represents an “ambiguous” expression whose children are each guarded by a different preprocessor configuration.

This representation can be constructed by modifying an LA(1) shift-reduce parser (most likely an LALR(1) [8] parser). A detailed discussion is beyond the scope of this paper; a more substantial treatment by the present authors is in preparation. The following method is oversimplified—its positioning of ambiguous nodes in the AST is often less than ideal—but it nevertheless conveys the gist of our technique.¹² The parser proceeds as usual until it reaches an `#ifdef` directive in the token stream. The parser clones itself so that a different copy of the parser can process each branch of the `#ifdef`. Each clone independently processes the tokens under its branch of the conditional, stopping when it is about to shift the token following the `#endif`. The clones are then compared for *equivalence*: If two or more clones are in the same state and have equivalent stack contents, those clones are merged into a single parser.¹³ When parsers are merged,

¹² Our technique is based on theory stemming from incremental parsing [4], although it appears that the application of the same essential ideas to preprocessing was independently discovered and implemented in [29].

¹³ Why is it safe to merge parsers under these conditions? In mathematical models of deterministic shift-reduce parsers, the transition relation between states is a *function* of the stack contents, the current state, and the remaining input: If all of these are identical among parsers, the parsers’ subsequent behaviors will be identical.

the topmost elements on their stacks are made to be children of an “ambiguous” node (as in Figure 1(c)), and this “ambiguous” node becomes the topmost element on the stack of the merged parser. After testing for equivalence and possibly merging parsers, each subsequent token is fed to all remaining clones, which shift the token and perform any reductions, stopping when they are prepared to shift the following token. The clones are again tested for equivalence, merged as necessary, and the cycle continues until only a single parser remains. In the worst case, this will happen at the end of the input.

An important implementation detail involves cloning parsers and testing for “equivalence” of parse stacks. When a parser is cloned, each clone should have an independent stack. However, there will be AST nodes present on the parser stack prior to cloning, and these nodes should remain pointer-identical among the clones’ stacks. Now, we can define two or more parser stacks to have “equivalent” contents when (1) the number of elements on each stack is the same, (2) the AST nodes on the top of the stacks all have the same type, and (3) all elements below the top on each stack are pointer-identical among stacks. This ensures that the merging parsers, as described above, will be viable.

Source Reproduction. Conditional completion and ambiguity control complicate source reproduction slightly, since some tokens may appear at several locations in a multiple-configuration AST even though they occur only once in the source text. Recall that, when a parser is cloned, any tokens already on the stack are pointer-identical among the clones; tokens beyond the `#endif` are fed to all clones, and, again, should be pointer-identical. Since the clones may place these tokens under different parents, it follows that these tokens may have *multiple* parents in the AST, i.e., the “multiple-configuration AST” is really a DAG. To reproduce the original source text, the conditional directives (`#if`, `#endif`, etc.) may be attached to tokens as whitetext, as before; during the traversal of the AST (DAG), a token’s text is printed only once.

The question is, should tokens with multiple parents be printed the first time they are traversed or the last time? Clearly, any shared tokens preceding the `#ifdef` should be printed the first time they are traversed; tokens following the `#endif` should be printed the last time. We can achieve this by simply *flagging* tokens following the `#endif` that are fed to multiple clones; during a source reproduction traversal, flagged tokens are printed the last time they are traversed (i.e., when they are traversed as a child of their rightmost parent), while all other tokens are printed the first time they are traversed.¹⁴

Multiple-Configuration Macro Expansion. Handling multiple preprocessor configurations means also handling the possibility that macros will have several possible expansions, depending on the configuration. Fortunately, this can be handled similarly to `#ifdefs`: The parser “forks” into several clones, each of which handles one expansion, then the clones synchronously handle tokens following the macro call, eventually merging back into a single parser.

¹⁴ Observe that a single flag will suffice regardless of the level of `#ifdef` nesting and number of parent nodes due to the lexical positioning of shared tokens.

5 Empirical Results

A rewritable AST generator has been implemented in Ludwig, a lexer and parser generator which is available for download [24]. Ludwig’s AST generator has been used in several projects; most notably, it has been used to generate the parser and rewritable AST in Photran [28], an open-source integrated development environment and refactoring tool for Fortran. Fortran 95 has an exorbitant amount of syntax—Photran’s grammar is nearly 2,500 lines long (a similar annotated grammar for Java 1.0 is just over 600 lines)—which has made it an ideal candidate for AST generation. At the time of writing, Photran contains 329 AST node classes comprising 33,081 lines of code; the AST base classes, parser, and semantic actions comprise another 25,909 lines. In total, then, its 2,500-line annotated grammar generates nearly 59,000 lines of code.

Photran’s parser was used to construct fully-concretized ASTs for 209 files from the source code for IBEAM [17], a massively parallel astrophysics framework based on the University of Chicago’s FLASH code [1]. The results are summarized in Table 1. Note that the source file sizes were positively skewed ($skew = 2.99$) but fairly well-correlated with the number of AST nodes ($r = 0.77$), as one would expect in Fortran. (This would not necessarily be the case in C, which uses preprocessing much more heavily. In general, the data in Table 1 are intended to be informative, not representative, as they depend on the Fortran language, Photran’s AST structure, and even IBEAM’s coding conventions.)

The third section of Table 1 is perhaps the most interesting: A rough, implementation-independent approximation of the size of an AST was computed by assuming that each interior node requires 32 bytes of overhead and each token requires 64 bytes. The *non-*

Table 1. Summary of data collected from 209 Photran ASTs. The first section of the table decomposes the nodes of the concretized ASTs by type. The second describes the partitioning of characters among tokens. The third estimates AST sizes as described in §5, and the fourth gives the size of the original source files in bytes.

Description	Median	Mean	St. Dev.	Min.	Max.
AST nodes	2,612	5,203	7,992	15	50,084
Interior nodes (N_I)	2,065 (79%)	3,931	6,198	8	41,003
Tokens (N_T)	558	1,272	1,840	7	11,871
Hidden	169 (6%)	409	615	0	4,099
Not hidden (N_N)	409 (16%)	863	1,232	7	7,772
Characters (with includes) (C)	4,861	9,663	12,953	33	81,373
in non-hidden tokens’ text (C_N)	1,606 (33%)	3,993	5,826	27	42,612
in hidden tokens’ text	223 (5%)	483	719	0	4,380
in whitetext	2,904 (60%)	5,187	6,884	6	46,521
Approximate AST size (bytes)					
Non-concretized AST	91,581	184,998	280,902	731	1,718,789
Transformation baseline	95,438	190,669	287,026	737	1,739,467
Concretized AST	108,780	216,858	325,216	737	1,942,877
Increase from non-concretized	20%	27%	16%	1%	98%
Increase from baseline	14%	15%	5%	0%	38%
Source file size (bytes)	4,114	7,926	10,891	33	69,472

concretized AST size is intended to approximate the size of an AST similar to one used in a compiler, one which contains neither hidden tokens nor whitetext; it is computed as $32N_I + 64N_N + C_N$ with variables defined according to Table 1. The *transformation baseline* assumes that the smallest possible AST is one in which every node has a fixed size, no hidden tokens are stored, and no text is stored (rather, each node contains an offset and length into a source file); to use this AST for source transformation, one would generally store the entire file (and any preprocessor-included files) in memory in addition to the AST, and so the total amount of memory required is $32N_I + 64N_N + C$. The *concretized AST* includes all source text in tokens and thus is computed as $32N_I + 64N_T + C$. The final two rows show the relative increase in memory requirements of a concretized AST over a non-concretized AST or the transformation baseline. One should note that the median increase in size due to concretization is relatively small—20% over a non-concretized AST and 14% over the transformation baseline—and even the largest concretized AST was estimated at less than 2 MB, a minimal demand on modern systems.

6 Related Work

There is an abundance of work on abstract syntax as well as work on handling the C preprocessor in the context of C and C++. The present work is distinguished by (1) its annotation-based approach for the definition of abstract syntax (as opposed to a more traditional declarative approach), (2) its language-agnostic handling of the C preprocessor, and (3) its focus on practical issues, including layout retention, customizability, and exposure as an API.

The Zephyr abstract syntax description language [36] is essentially a declarative language for “tree-like data structures” and resembles declarations of algebraic data types (à la ML). Wile [37] advocates a particular grammar notation (WBNF) which makes some aspects of abstract syntax (e.g., iteration, optionality, precedence, and nesting) more explicit and suggests a heuristic process by which Yacc grammars can be converted and abstract syntax derived automatically. Among existing tools, ANTLR [27], LPG [23], and JavaCC/JJTree [20] all include AST generators; LPG’s is derived directly from the concrete syntax, while ANTLR and JJTree rely on declarative specifications embedded in the grammar.

Our approach differs in many ways from all of these. For example, Zephyr is not integrated with a parser generator; ANTLR’s ASTs allow a limited form of source rewriting [27, Ch. 9] but have a dramatically different structure than the ASTs we describe; and no existing AST generator can generate ASTs for C-preprocessed sources. However, the most prominent distinction of the present work is its annotation-based approach. Zephyr, ANTLR, and JJTree require the user to fully specify an abstract syntax. LPG constructs a primitive abstract syntax from the concrete syntax automatically. In contrast, our approach allows a *default* abstract syntax to be constructed from the concrete syntax and subsequently *refined* using annotations. This places less of an annotation burden on the programmer than requiring a fully explicit abstract syntax definition while simultaneously allowing more flexibility than a fully-inferred definition. One drawback of the annotation-based approach is that the structure of AST nodes is not immediately

obvious, particularly when (*inline*) annotations are used; to remedy this, in Ludwig, we are developing a GUI which allows the user to view AST node structures as the grammar is being annotated.

Our heuristic for attaching whitetext to tokens in order to facilitate rewriting also appears to be new, although the fundamental idea of including whitetext in syntax trees appears elsewhere. Sellink and Verhoef [30] suggest placing whitespace and comments into a parse tree by (automatically) modifying the grammar to include a “layout” non-terminal prior to each occurrence of a token; Lämmel [21] includes such information as annotations in the parse tree.

To our knowledge, our work is also the first to treat language-independent handling of C-preprocessed code at length. An empirical confirmation of the importance of the C-preprocessor is [11]. The conditional completion problem is defined in [14], and a solution similar to ours appears in [29]. Garrido [13,14,15] treats refactoring C-preprocessed code in detail; [25] takes a different approach, requiring a semi-automated replacement of C preprocessor directives with a syntactically embedded macro language.

Finally, inasmuch as the present work addresses the topic of language-independent refactoring (and language-independent program transformation), one should note that there is a great body of literature in these areas, and commonality extends beyond just syntactic and lexical issues. For example, Lämmel [22] suggests that refactorings have both language-independent and language-dependent components and can be built by parameterizing a transformation with language specifics, while Mens *et al.* [26] investigate the viability of graph rewriting as a formalism for specifying refactorings.

7 Conclusions

We have proposed six *grammar annotations*—omission, labeling, Boolean access, list formation, inlining, and superclass formation—that allow an abstract syntax for a language to be defined based on its concrete syntax. A tool can use such an annotated grammar to generate both a parser and a rewritable AST. *Concretizing* the AST allows it to preserve the formatting of the original code even after rewriting. Furthermore, an AST generator based on our method can couple the generated parser/AST-builder with a *pseudo-preprocessor* to allow representation and manipulation of preprocessed code. Our AST generator has been implemented in a tool called Ludwig [24] and has been used to generate the rewritable AST in a refactoring tool for Fortran; the annotated grammar was more than an order of magnitude smaller than the generated code, and the overhead of concretizing ASTs was very reasonable.

Much future work is possible. The present authors are working on a complete description and formalization of the AST generation algorithm: It is easy to develop an annotated grammar that does not “make sense” (for example, if a node indirectly inlines itself, or if a field has an ambiguous type, or if a node is simultaneously declared as a superclass and a list), so an AST generator must be able to detect errors and supply the user with an informative message rather than failing or generating invalid code. Grammars for mainstream programming languages other than Fortran should be developed in order to better assess our choice of annotations and the overhead of concretization. Constructing multiple-configuration ASTs efficiently using recursive descent parsers

remains an open problem. To our knowledge, no one has addressed refactoring in the presence of other preprocessors (e.g., m5) in detail; it is intriguing to consider the possibility that a programmer could chain together an arbitrary sequence of pseudo-preprocessors for refactoring, just as one would chain together several tools prior to compilation in a Makefile. Most importantly, very few programming languages currently have production-grade refactoring tools available; we hope that our work will enable researchers and practitioners to implement and investigate refactorings for a variety of languages, allowing programmers using those languages to see the same productivity gains that others already have.

Acknowledgement

This work was supported by the United States Department of Energy under Contract No. DE-FG02-06ER25752 and by the National Science Foundation under NSF award #0725070. The authors would like to thank the anonymous referees and the members of the UIUC Software Engineering Seminar, particularly Darko Marinov and Danny Dig, for their detailed feedback.

References

1. ASC Center for Astrophysical Thermonuclear Flashes, <http://www.flash.uchicago.edu>
2. Bečička, J., Hřebek, P., Zajac, P.: (Sun Microsystems): Using Java 6 compiler as a refactoring and an analysis engine. In: 1st Workshop on Refactoring Tools (2007)
3. Bowdidge, R.: (Apple Computer): Personal communication
4. Celentano, A.: Incremental LR parsers. *Acta Inf.* 10, 307–321 (1978)
5. Cordy, J.: The TXL source transformation language. *Sci. Comp. Prog.* 61, 190–210 (2006)
6. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: *Introduction to Algorithms*. 2/e. MIT Press, Cambridge (2001)
7. De Jonge, M., Visser, E., Visser, J.: XT: A bundle of program transformation tools. In: *Proc. Lang. Descriptions, Tools and Applications 2001*. *Elec. Notes in Theoretical Comp. Sci.*, vol. 44, pp. 211–218 (2001)
8. DeRemer, F.: *Practical translators for LR(k) languages*. PhD thesis, MIT (1969)
9. Eclipse C/C++ Development Tools, <http://www.eclipse.org/cdt>
10. Eclipse Java Development Tools, <http://www.eclipse.org/jdt>
11. Ernst, M., Badros, G., Notkin, D.: An empirical analysis of C preprocessor use. *IEEE Trans. on Software Eng.* 28, 1146–1170 (2002)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, Reading (1994)
13. Garrido, A., Johnson, R.: Challenges of refactoring C programs. In: *Proc. Intl. Workshop on Principles of Software Evolution*, pp. 6–14 (2002)
14. Garrido, A.: *Program refactoring in the presence of preprocessor directives*. PhD thesis, University of Illinois at Urbana-Champaign (2005)
15. Garrido, A., Johnson, R.: Refactoring C with conditional compilation. In: *Proc. 18th IEEE Intl. Conf. on Automated Software Eng.*, pp. 323–326 (2003)
16. Hopcroft, J., Motwani, R., Ulman, J.: *Introduction to automata theory, languages, and computation*. 2/e. Addison-Wesley, Reading (2001)

17. IBEAM, <http://www.ibeam.org>
18. ISO/IEC 9899:1999: Programming Languages – C
19. ISO/IEC 14882:2003: Programming Languages – C++
20. JJTree, <https://javacc.dev.java.net/doc/JJTree.html>
21. Kort, J., Lämmel, R.: Parse-tree annotations meet re-engineering concerns. In: Proc. Source Code Analysis and Manipulation, pp. 161–172 (2003)
22. Lämmel, R.: Towards generic refactoring. In: Proc. ACM SIGPLAN Workshop on Rule-Based Prog., pp. 15–28 (2002)
23. LPG, <http://sourceforge.net/projects/lpg/>
24. Ludwig, <http://jeff.over.bz/software/ludwig>
25. McCloskey, B., Brewer, E.: ASTEC: A new approach to refactoring C. In: Proc. 13th ACM SIGSOFT Intl. Symp. Found. Software Eng., pp. 21–30 (2005)
26. Mens, T., Van Eetvelde, N., Demeyer, S., Janssens, D.: Formalizing refactorings with graph transformations. *J. Software Maint. and Evolution* 17, 247–276 (2005)
27. Parr, T.: The definitive ANTLR reference: Building domain-specific languages. Pragmatic Bookshelf, Raleigh (2007)
28. Photran, <http://www.eclipse.org/photran>
29. Platoff, M., Wagner, M., Camaratta, J.: An integrated program representation and toolkit for the maintenance of C programs. In: Proc. Conf. Software Maint., pp. 129–137 (1991)
30. Sellink, M., Verhoef, C.: Scaffolding for software renovation. In: Proc. Conf. Software Maint. Reeng., pp. 161–172 (2000)
31. Stallman, R., Weinberg, Z.: The C preprocessor, <http://gcc.gnu.org/onlinedocs/cpp.pdf>
32. Tomita, M.: Generalized LR parsing. Springer, Heidelberg (1991)
33. 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)
34. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 357–361. Springer, Heidelberg (2001)
35. Vlissides, J.: Pattern hatching: Design patterns applied. Addison-Wesley, Reading (1998)
36. Wang, D., Appel, A., Korn, J., Serra, C.: The Zephyr abstract syntax description language. In: USENIX Workshop on Domain-Specific Languages (1997)
37. Wile, D.: Abstract syntax from concrete syntax. In: Proc. 19th Intl. Conf. on Software Eng., pp. 472–480 (1997)

Systematic Usage of Embedded Modelling Languages in Automated Model Transformation Chains

Mathias Fritzsche^{1,3}, Jendrik Johannes², Uwe Aßmann², Simon Mitschke^{1,2},
Wasif Gilani¹, Ivor Spence³, John Brown³, and Peter Kilpatrick³

¹ SAP Research CEC Belfast
Shore Road, BT370QB Newtownabbey
United Kingdom

mathias.fritzsche@sap.com, simon.mitschke@sap.com, wasif.gilani@sap.com

² Technische Universität Dresden
D-01062, Dresden, Germany

jendrik.johannes@tu-dresden.de, uwe.assmann@tu-dresden.de

³ Queen's University Belfast
University Road, Belfast BT7 1NN
United Kingdom

i.spence@qub.ac.uk, tj.Brown@qub.ac.uk, p.kilpatrick@qub.ac.uk

Abstract. Annotation of programs using embedded Domain-Specific Languages (embedded DSLs), such as the program annotation facility for the Java programming language, is a well-known practice in computer science. In this paper we argue for and propose a specialized approach for the usage of embedded Domain-Specific Modelling Languages (embedded DSMLs) in Model-Driven Engineering (MDE) processes that in particular supports automated many-step model transformation chains. It can happen that information defined at some point, using an embedded DSML, is not required in the next immediate transformation step, but in a later one. We propose a new approach of model annotation enabling flexible many-step transformation chains. The approach utilizes a combination of embedded DSMLs, trace models and a megamodel. We demonstrate our approach based on an example MDE process and an industrial case study.

1 Introduction

The usage of Domain-Specific Modelling Languages (DSMLs) in Model-Driven Engineering (MDE) is increasing in popularity. Traditionally, it is a common practice to define and use Domain-Specific Languages (DSLs) by embedding them into other languages [1]. Bravenboer et al [2] proposed a GUI definition language embedded into Java. Another common example is the usage of SQL statements as Strings in Java or PHP, which is effectively language embedding. While this is in general not a new idea, new challenges emerge when this practice is applied for DSMLs in the MDE domain.

The distinction between DSMLs and traditional DSLs is not formally defined. There are, however, certain aspects which tend to be different. For example, the metalanguages used to define language syntax tend to be different; DSMLs are often defined through MOF-like [3] metamodeling languages while traditional DSLs are often defined through Extended Backus-Naur Form (EBNF)-like grammars. The syntax of DSMLs tends to be more graphical, while traditional DSLs more often come with textual syntax. These are only some examples of how DSMLs and traditional DSLs tend to differ in general.

In this paper we focus on embedded DSMLs and how semantic definition and compilation of them differs compared with traditional embedded DSLs. We identify problems that arise from these differences and propose solutions to tackle the identified problems.

The difference between DSMLs and traditional DSLs is related to the difference between programming and modelling. Programs can be directly interpreted or compiled into an executable system. Models—in the sense of MDE—on the contrary are abstract descriptions that need to be enriched with additional information to obtain an executable system. Therefore, traditional DSLs—that are special-purpose programming languages—require fully defined executable semantics to be usable. DSMLs on the contrary can well be used without having full executable semantics defined. There are different approaches to define the semantics for a traditional embedded DSL:

1. **Interpreter written in the host language:** It is a common practice, for instance, to define SQL statements as Strings in Java. Those are then preserved during compilation and interpreted at runtime by a Java method. The semantics are then defined inside the host language that embeds the DSL.
2. **Compiling to the host language:** Another approach is to embed a DSL into a host language by extending the host language with new syntax (e.g., [2]). The new constructs are then compiled down to constructs of the original language.
3. **Compiling embedded and host language to a common base language:** Quite similar is the approach to extend the host language, but then compile both host and embedded into a common base language. In AspectJ for instance Java classes and AspectJ aspects are both compiled to and integrated in Java byte code.

If we look at embedding DSMLs in modelling languages in a first and using the embedded information in a second step, compilation steps might be involved—which are called *model transformations*. The most prominent example is the application of a profile in UML. A UML profile defines a DSML that can be injected into UML in a standard way. The information defined in the profile application (that is in the embedded DSML) is then interpreted in a model transformation. This transformation can also produce another UML model (e.g., Platform Independent Models (PIM) to Platform Specific Models (PSM) in MDA [4]) that resolves the information in the profile into ordinary UML elements.

This is in fact a compilation to the host language (see 2 above). The transformation can also produce instances of another modelling language which is similar to compiling host and embedded language into a common base language (3 above).

Consider now the case that the result of the compilation is, after further enrichment, again automatically compiled (i.e., transformed) and so on. Such an automated many-step transformation chain is more systematic than a single transformation, e.g. to realize modularity or, in other words, to achieve separation of concerns [5] for complex transformations. In such an automated many-step transformation chain, it might well happen that information defined in an embedded DSML at some point is not required in the next compilation step, but in a later one. Since the information is required at some point, it needs to be preserved throughout the transformation chain. This has the drawback that the information that is not of interest in a particular intermediate model has to be put directly into the model anyway to be picked-up by the next transformation.

This is, however, not desired in MDE, because the target languages of transformation are actually DSMLs should only provide constructs that are of interest for the particular domain they are specified for.

We identified this problem in a concrete MDE process called Model-Driven Performance Engineering (MDPE) [6]. The process defines an automated many-step transformation chain and requires the *annotation* of additional information at certain points in the process that is only evaluated by a transformation later in the chain and not of interest in intermediate models. As we will see, the languages in which the annotations are defined are in fact embedded DSMLs, where the language of the annotated models is the host DSML. Profiling in UML, for instance, is a specific type of annotation where UML is the host language and the annotation language (i.e., the profile) is the embedded DSML.

In this paper we propose a methodology that traces any information defined in any embedded DSML in a many-step transformation chain in such a way that it can be accessed at any point in the chain when it is needed. The approach can be applied for automated transformation chains where no human interaction is required. For this, we unify the way in which DSMLs for annotations are defined. Using this methodology, transformations and intermediate models do not need to be polluted by handling locally unnecessary metadata.

The rest of the paper is structured as follows: Section 2.2 introduces MDPE as an example of a process and a use case applying that process. Section 3 discusses the problems that occur in the application of the MDPE process with respect to handling annotations defined in embedded DSLs. In Section 4 we propose a system architecture based on MDE technologies that systematises the handling of annotations and unifies the definition and usage of embedded DSMLs for annotations. Section 5 gives details of our implementation of the architecture and discusses its advantages based on experience gained so far. In Section 6 we give pointers to related work and conclude.

2 Scenario: Language Engineering in Model-Driven Performance Engineering

In this section we introduce Model-Driven Performance Engineering (MDPE) as one example of MDE process implementing an automated many-step transformation chain.

MDPE has been defined in order to enable performance engineering based on development models¹ and additional performance related data [6]. The process therefore provides performance-related² decision support to business domain experts [7,8], i.e. support decisions for resource scheduling problems via business simulations.

In the following subsection we give an overview of the industrial case study that we employed to gain experiences with the MDPE process. A brief overview of the stepwise process follows in subsection 2.2.

2.1 Use Case

In our special case, a business domain expert uses SAP NetWeaver Business Process Management (aka SAP NetWeaver BPM) to define business process extensions by applying MDE concepts. A Business Process Modelling Notation [9] based tool called Galaxy, announced in May 2008, is intended to be used for this purpose[10]. We assume the Business Process Modelling Notation as an example for a DSML.

In [11], we presented a SAP NetWeaver BPM based case study called **Wine Under Special Treatment (xWURST)**. There, an already running back-end business process called “Sales Order Processing” is extended with a so-called front-end process [11] to meet the specific needs of a wine seller. The behaviour of both, the back-end as well as the front-end process, is assumed to be available as a model. For the back-end processes, SAP proprietary models are used whereas front-end processes are modelled in BPMN using the previously mentioned SAP NetWeaver BPM (see Figure 1). We used this case study to gain firsthand experience with the approach described in this paper.

Within the case study, the back-end process is extended so that it is possible to add as compensation a free bottle of wine only to orders of those customers who notified a quality issue for their previous order. The decision about which wine will be added for free has to be taken manually by a wine specialist based on the wine rating and the customer’s purchase history over the last 12 months. Additionally, the selection has to be approved by a manager.

We claim that the business domain expert needs support from the business performance perspective, since there are several steps within the business applications which have to be processed by human resources. The decision about

¹ We use the term *development model* to distinguish between models as development artefacts and formal *performance models*.

² Performance is defined as the “degree to which an application or component meets its objectives for timeliness”.

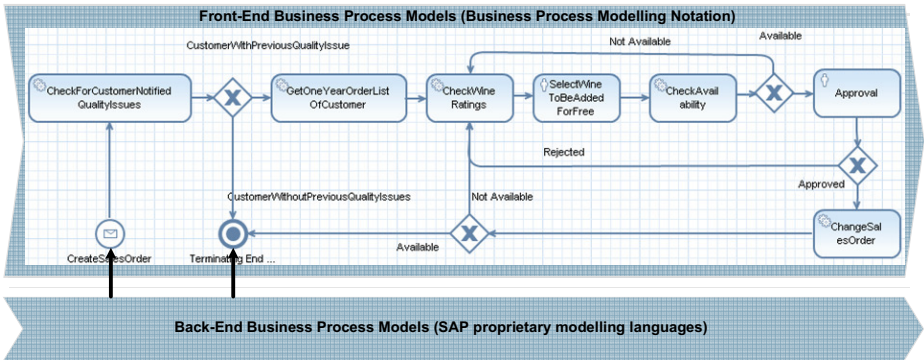


Fig. 1. Extension of an already existing back-end business process modelled with the Business Process Modelling Notation (BPMN)

which human resource performs which step in a business process is not trivial to make. In addition, the former difficulty is further increased by the flexibility introduced by the possibility of extending the back-end processes with front-end processes by a domain expert, with no performance related skills. This can dramatically change the behaviour of already running and stable processes.

Therefore, the necessity of providing performance related decision support for a domain expert, i.e. a business domain expert, is obvious. Such support can be provided through an MDE process that involves multiple modelling languages—including embedded DSMLs for annotations—and that is realised by various transformation, annotation, and tracing tools. MDPE is such a process, and is described in the following subsection.

2.2 Stepwise Transformation for Performance Engineering

With the MDPE process [6,11], we implemented an extensible approach which can be applied to different use cases, not only to the one presented in the previous subsection. A brief description of the latest version of the MDPE process is given in order to provide an example for the need of a systematic approach for model annotation with embedded DSMLs in transformation chains.

Most parts of the MDPE process are implemented within the *MDPE Workbench* (see figure 2). This workbench is developed based on Eclipse and the Eclipse Modelling Framework (EMF) in order to provide a widely usable framework. However, we developed a *MDPE Workbench Adapter4SAP* in order to use the MDPE Workbench in integration with the SAP NetWeaver BPM tooling, which is mainly based on SAP's proprietary Modelling Infrastructure (MOIN) [12].

Within the MDPE process, we do not directly transform *development models*, such as the model shown in figure 1, into *Tool Specific Performance Models (TSPMs)* but stepwise via intermediary *Tool Independent Performance Models (TIPMs)*. Thus, we decided to implement our transformation in a modular way. This enables us to deal not only with other proprietary modelling languages, such

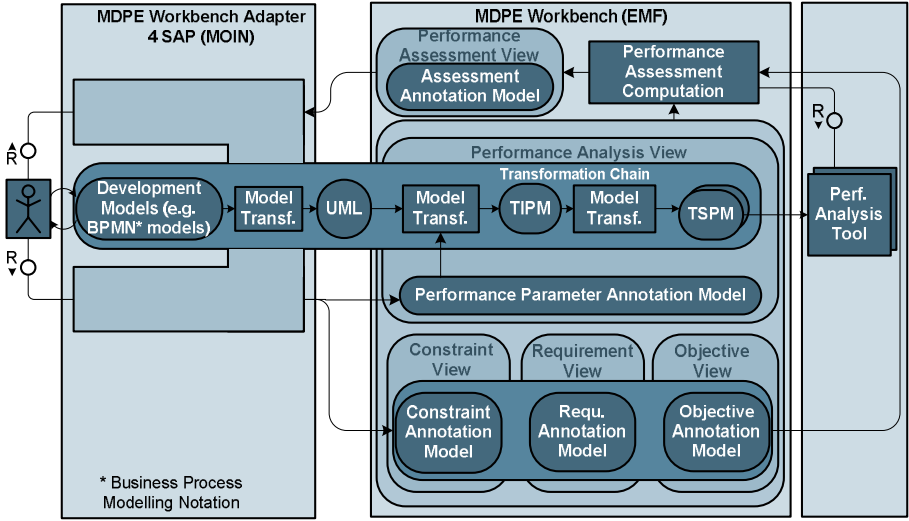


Fig. 2. Model-Driven Performance Engineering as block diagram ([14])

as the languages for the purpose of back-end process modelling, but also well-known modelling languages, such as UML. Additionally, tool independence is of high value for business software vendors, such as SAP, in order to be independent of one specific simulation tool, such as the one employed in the case study called AnyLogic [13]. The TIPM metamodel represents behaviour information, available resources and consumption of resources.

However, the current implementation of the so-called *MDPE Workbench Adapter4SAP* is not transforming proprietary models directly to the TIPM but to UML models in order to build a bridge between the TIPM and SAP proprietary models. We use this approach as it enables SAP to share proprietary models with partners of research projects without publishing the metamodels. Additionally, we are able to modify the content of the proprietary models slightly within a transformation to UML models in order to avoid publishing confidential details of the implemented business processes.

It is obvious that we are required to annotate *performance parameters* (see Figure 2), such as resource demands, branch probabilities, and factors due to contention for resources in order to transform our development models stepwise so as to prepare them for performance simulation.

In the use case, introduced in the previous subsection, we are required to annotate the average resource demands. Annotations have to be done in terms of employee time, for each manually processed step and the resource mapping, in particular, how many employees are working on which manual step in our behaviour models. This data can be extracted from the past history if the process has already been productive, and assumed values have to be annotated for newly defined parts of a business process.

In [7] we described the need to model not only the performance parameters, but also *Modification Constraints*, *Performance Requirements* and *Performance Objectives* in order to provide performance related decision support for business domain experts who generally lack performance expertise. Therefore, we additionally need to annotate development models with this kind of information, as shown in Figure 2. The information is used in order to compute the *Performance Assessment Computation* view to be presented to the domain expert. Based on the annotated information, the Performance Assessment View can, for instance, contain a proposal guiding a business domain expert to the optimal resource mapping or optimized design solution.

In [8] we described the tracing of simple simulation results back to the development models by using *Trace Models*, storing the information about which source element(s) is mapped to which target element(s) via a model transformation. We use the same approach in order to trace the more beneficial Performance Assessment View.

The additional step in the chain (transforming SAP models to UML), but also the modelling of Modification Constraints, Performance Requirements and Performance Objectives, which was not considered in [8], introduces new challenges with respect to annotation and tracing of the annotated information in the chain. The following section discusses these challenges.

3 Problem Motivation

It is obvious that we are required to annotate additional data, such as performance parameters (cf. Figure 2) to the business process models (cf. Figure 1), as described in the previous section.

The interesting point about the annotated information is that it is accessed only after a number of automated transformation steps. It can be seen in Figure 2 that we use the Performance Parameter information initially in the second transformation step to generate a TIPM. The annotated Modification Constraints, Performance Requirements, and Performance Objectives are initially used for the Assessment Computation (cf. upper part of Figure 2) after the TIPM has been generated. However, all annotations are performed based on the development models as the business process modeller does not wish to access models which are pure performance analysis models, such as the TIPM. For our initial UML-based implementation of a preliminary version of the transformation chain we used UML Profiles as a UML embedded DSML for the annotation of the Modification Constraints, Performance Requirements and Performance Objectives, as they are well supported by UML-based tools [7].

However, we identified a number of shortcomings with this approach when we started to work with a longer transformation chain and with modelling languages other than UML.

Non-UML modelling languages, such as models conforming to the Business Process Modelling Notation and SAP proprietary languages used for our current case study (see subsection 2.1), do not normally support a profiling mechanism which is an extension mechanism on the meta-level.

However, modifying the metamodel of proprietary models is not a systematic solution as it would require polluting the host DSML with information that is not domain-specific. This turns out to be a problem especially if models are provided by a third party. This is true for back-end business processes in our case study (cf. Subsection 2.1), which may partly be provided by SAP and partly by a third party. In such cases we might not be able to extend the modelling language of the third party process, and, therefore, cannot annotate the third party models if they do not provide any extension mechanism.

In addition, transformations that do not require access to the annotated information, such as the transformation from development models to UML (cf. left side of Figure 2), obviously should not be polluted with the annotated information, since the annotated information is only needed in the UML to TIPM transformation. Alternatively, it would be possible to use the originally extended development model as an additional input for the UML to TIPM transformation. However, both options are not effective as in both cases more data is used as an input for a transformation than required. Hence, metamodel extensions, such as provided by UML profiles, do not support chains of transformations systematically.

Therefore, an approach that enables host model extensions without manipulating them in an MDE process where the metamodels do not support systematic model extensions and the metamodel can or should not be changed is needed here. Furthermore, the approach needs to explicitly support tracing of annotated information in transformation chains such that information can be accessed directly only when it is needed and not earlier for preservation reasons. In the next section we define such an approach.

4 Proposed Solution

Figure 3 gives an overview of our proposed architecture enabling systematic annotations for model transformation chains, namely the MDPE transformation chain in our case.

Our approach takes the following three model types into account in order to annotate development models (cf. left side of Figure 3):

1. **Annotation Models**, as described in [15,16], are instance of a specific isolated metamodel. Such a metamodel effectively defines an embedded DSML and consists of two parts: one for the annotation itself, and one for the information about how annotations are composed with the original host models, such as the development models in the MDPE case. The first (annotation) part defines the domain of the embedded DSML—for example, performance parameters or performance objectives. The second (weaving) part, defines how the language is embedded into the host language—that is, which annotations are linked to which elements in the development models. Due to the fact that this linkage information is included in the Annotation Model, it is not necessary to pollute the original development models with this data or to extend their metamodel(s).

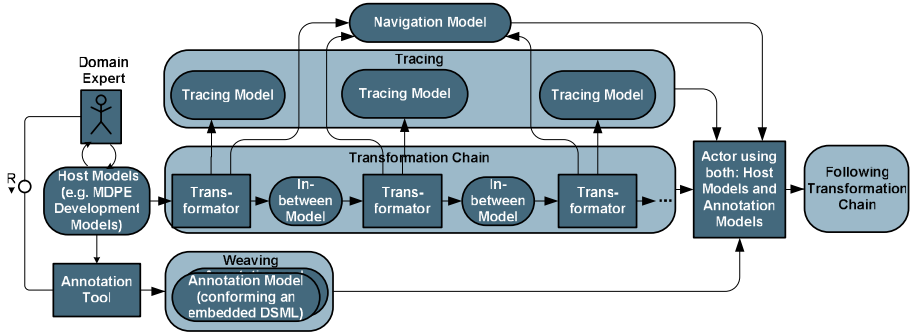


Fig. 3. Proposed Architecture

2. **Tracing Models** as described in [17,18,19] contain information about which source model elements have been transformed to which target model elements in a transformation chain (cf. centre of Figure 3). Hence, tracing models contain associations between models conforming to two different meta-models. We use tracing models in our approach as input for tools which are actually using the annotated information. The trace model is required in order to know at any position p in a model transformation chain, which model element(s) el_p are associated with which source model element(s) el_{source} . This is necessary in order to know which annotation is (indirectly) linked with which model element el_p .
3. A **Navigation Model** is used to associate models in a transformation chain with the related tracing models, such as described in [20]. This is required since we assume that the models in a transformation chain are non-changeable and therefore cannot be polluted with the tracing model linkage information.

5 Implementation

A description of an implementation of the concepts mentioned in the previous section is provided in this section.

5.1 Annotation Models

The annotation of models in transformation chains, such as the development models shown in Figure 1, is done based on Annotation Models as briefed in the previous section.

We have developed a generic approach which is, for instance, usable for the annotation of development models in the MDPE process. For this process, we are required to support a number of different embedded DSMLs used at different steps in the MDPE transformation chain as shown in Section 2.

It can be seen in the upper part of Figure 4 that our approach inherits from the weaving metamodel *MWCore* provided by the ATLAS group [21]. We selected this metamodel due to its integration in the Eclipse framework.

In comparison to other model annotation approaches based on the MWCORE metamodel ([15] and [16]), our annotation approach is based on a generic metamodel for model annotations, the *Annotation Meta Model* (AMM, see middle part of Figure 4). It refines the meta-class *WLink* from the MWCORE metamodel with the meta-class *Annotation*. The *WLink* class is indirectly associated with an attribute “ref” of type String that is used to represent references into our proprietary modelling repository MOIN. Hence, the Annotation model elements represent the weaving link to the host DSML elements, which can be defined in any kind of modelling repository, which includes the SAP proprietary MOIN repository for our MDPE case. Additionally, an Annotation contains references to *AnnotationProperties* which represent the annotated information to the source model elements.

An AnnotationProperty can be further refined for the specific needs of a specific annotation metamodel (that is a specific embedded DSML). In order to well integrate our work in the Eclipse workbench, the extension of the Annotation Meta Model can be done by implementing an Eclipse Extension-Point which is provided by the plug-in implementing the Annotation Meta Model.

This architecture of loose coupling is employed by our generic editor for model annotations in order to support navigation through a specific annotation metamodel. Thus, based on the Annotation Meta Model our generic editor is usable for all kinds of metamodels which are extending the Annotation Meta Model, and is integrated in the Eclipse Workbench.

Additionally, the generic editor currently allows annotation of all kinds of AnnotationProperties to all kinds of model elements in the MDPE development models. In future, we anticipate to configure this mapping in a separate constraint model. However, in our current implementation we have hard coded the required constraints.

Additionally, we are able (through an extension point) to refine some classes of our generic editor in order to make it even more suitable for users. Figure 5 depicts the user interface of our editor for the Business Performance Annotation Meta Model, as one embedded DSML example.

The metamodel of this embedded DSML is depicted by the package called “BPAMM” at the lower part of Figure 4. It enables the specification of multiple business *Scenarios* in order to separate multiple cases for performance simulation, for instance, different locations of the Wine Seller company, introduced in subsection 2.1, such as “LocationChicago” and “LocationDenver” (see Figure 5). A business scenario is additionally related with a number of *Workloads*, *ExecutionPaths* and *ResourceUsages*.

Additionally, *Resources* can be defined which are aligned with a *ResourceResponsibility*. We use this in order to annotate a BusinessResource, such as the human resource “WineSpecialist.RayBiggs”, to a development model in the MDPE transformation chain (see figure 1). For each BusinessResource, one or multiple *ResourceResponsibilities* have to be defined in order to specify operation times for different Scenarios in a business. It is, for instance, possible to specify an operation time of “8_HoursPerWorkingDay” for the “WineSpecialist.RayBiggs”.

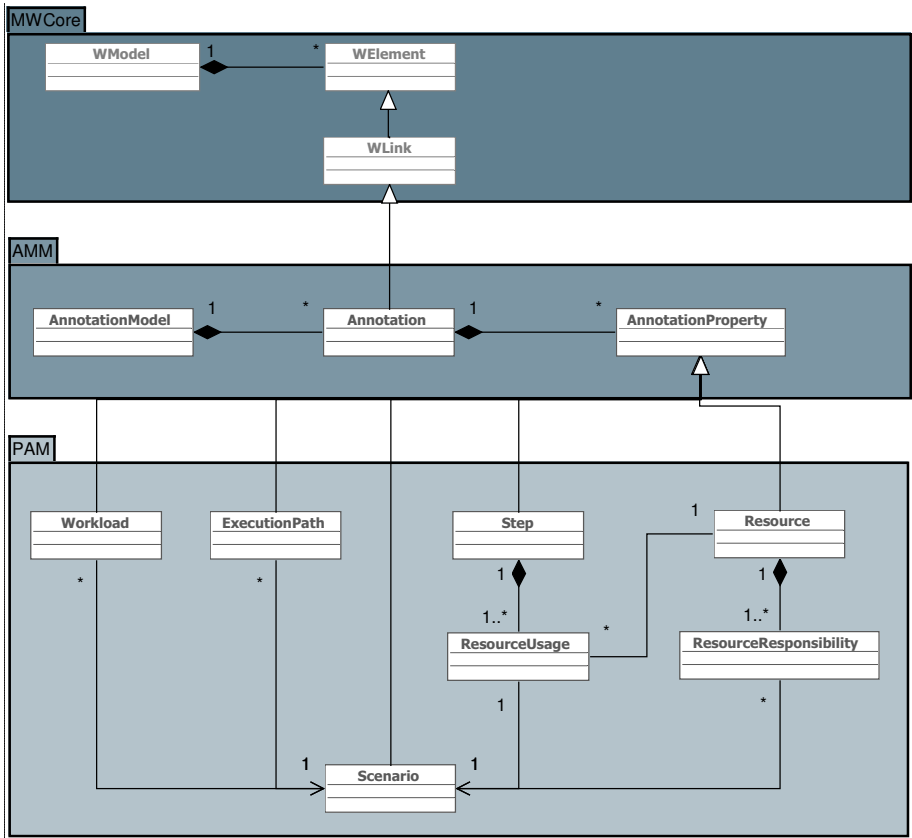


Fig. 4. Example for an embedded DSML used for the MDPE process (simplified metamodel)

The annotation of an *ExecutionPath* enables annotation of a probability based on a modelled behaviour as depicted by Figure 1. In particular, it is possible to model how often a specific path in a business process is executed in case there are more than one possibilities. We used this in our case study (see subsection 2.1), for instance, to specify that 10 percent of all approvals regarding which wine is added for free are rejected by the manager of the wine specialist.

BusinessSteps are used to annotate actions in our development models with resource demands and to associate them to a *Resource*. As can be seen by the screenshot in Figure 5, the manual action “SelectWineToBeAddedForFree” requires 5 minutes of the resource “WineSpecialist_RayBiggs” in “Chicago”.

5.2 Tracing Models

For the tracing issue we used the tracing metamodel by the ATLAS group [18]. There are other tracing metamodels available as well, such as [17]. However,

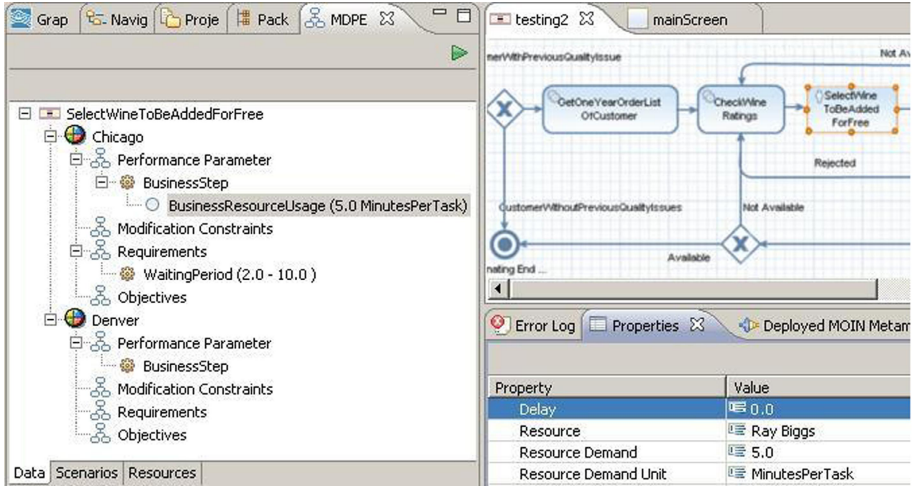


Fig. 5. Annotated action “SelectWineToBeAddedForFree”

for our MDPE case we have selected this metamodel since the ATLAS group also developed the Higher-Order Transformation mechanism [22,18] that we have employed for most transformations in our model chain. Higher-Order Transformations are transformations that are used to transform one transformation A to a new transformation A^* . The approach enables us to generate tracing models with minimal additional effort in our MDPE process, as presented in [8]. Higher-Order Transformations can be integrated in a batch-job as described by Jouault [18] in order to automatically extend the existing transformations in a way that they additionally output tracing models. This approach is used in [18] to automatically extend rules, for instance, within ATL [23] transformations, with code for additional information generation. This code creates a tracing model when the transformation is executed. Hence, tracing is achieved with minimal additional effort via Higher-Order Transformation.

5.3 Application of a Megamodel as Navigation Model

Based on the MDPE process, we experienced the requirement to be able to navigate between the modelling artifacts by taking their interrelationships into account. In particular, navigation from models in the transformation chain to their related trace models is required in order to know at any step in a model transformation chain which trace model is related to which model in the chain in order to retrieve the corresponding annotation information.

To our knowledge, there is just one approach available fulfilling this requirement called Megamodelling [20]. A megamodel represents *Models* (see Figure 6) and their *Relationships* (see Figure 6). Our current implementation defines the megamodel [20], and add not only the different MDPE models, i.e. the development models (see Figure 1), and the generated UML model, the TIPM, the

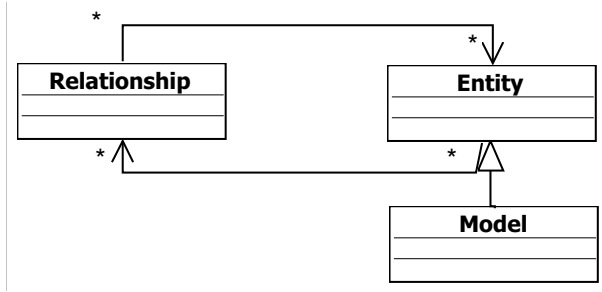


Fig. 6. Simplified metamodel of the megamodel defined in [20]

AnyLogic Simulation model and the trace models, but also Relationships among them, to it.

5.4 Use of Transformation Results and Annotated Information

Finally, there is a step within or after a transformation chain where the annotated information is required to be computed. A description about how this computation is done, for instance, for the MDPE case, introduced in subsection 2.1, follows.

Use of an embedded DSML for the UML to TIPM transformation in the MDPE process: Due to the tracing models and the megamodel, mentioned in the previous subsections, the transformation from UML to TIPM can compute which UML model element is associated with which model element in an initial MDPE development model (e.g., a Business Process Modelling Notation model). Therefore, the annotations, which are done on, for instance, business process models in the MDPE process, can directly be associated with UML model elements.

This association is computed in an imperative rule in our UML to TIPM transformation code. In order to make that imperative rule work, a trace model is required as an input for our UML to TIPM transformation. This trace model specifies the direct trace links between the development model elements and the UML model used as an input for the transformation. In the case of the UML to TIPM transformation, we simply use the trace model between development models of MDPE and UML generated with the help of a Higher-Order Transformation.

It is obvious that our approach is only useful in the case related target model element(s) are available for the annotated host model element(s) after the execution of an automated model transformation chain. If this is not the case then the domain expert needs to be informed by the tooling we provide since the annotation he did was useless for his purpose.

Use of embedded DSMLs for Performance Assessment Computation in the MDPE process: A second step where annotated information is used

is after the *Assessment Computation Actor* (see Figure 2). At this step the generated TIPM as well as the annotated Modification Constraints, Performance Requirements and Performance Objectives are required as input. This input is used to compute the Performance Assessment View, such as the computation, etc., of an optimal configuration, as described in [7]. For instance, we wanted to compute the optimal number of resources for the departments involved in the back-end process used in our case study - Sell-From-Stock-Scenario, which was extended with two additional manual steps to meet the specific needs of a wine seller (see subsection 2.1).

The Modification Constraints, Performance Requirements and Objectives are annotated on the original development models.

However, the Performance Assessment Computation is based on the TIPM, which is simulated using the Anylogic tool as described before. This Performance Assessment Computation needs knowledge about which model elements in the TIPM are associated with the Modification Constraints, Performance Requirements and Objectives.

The megamodel enables us to navigate to the trace model between UML and TIPM and the trace model between our TIPM development models and UML, which are then used in order to generate a new trace model that directly links proprietary and TIPM model elements. This is then utilized by the Performance Assessment Computation in order to relate TIPM model elements with model annotations.

Concluding, in many-step transformation chains, where a number of trace models are required as input, we need the megamodel to be able to navigate from models in a transformation chain to their trace models generated from the Higher-Order Transformations. This is needed to generate a direct trace model from, for instance, the models used as input for transformation step one to models used as output of transformation step three, such as in the MDPE case. Therefore, by making use of megamodel we are able to apply our approach even for longer transformation chains.

6 Related Work

While the usage of DSMLs in MDE is commonly seen as a good thing, there are many unresolved issues when it comes to semantics and the interoperability of DSMLs. Often DSMLs only integrate “somehow” by model transformations. More systematic approaches, like [24,25], propose more formal semantic integration of DSMLs. As the interest in this area increases, dedicated events emerge just now [26].

On the other hand, model annotations, and in particular UML profiles, are often used for the purpose of defining embedded DSMLs (also called language extensions) [27]. There is also an awareness that these kind of embedded languages have different properties than languages defined from scratch, but there is no common understanding what exactly the (dis)advantages of one or the other are in which scenario.

We are currently not aware of any work that explicitly handles model annotations as DSMLs and regards their specific properties in the context of many-step model transformation chains. We believe that this is an important issue worth investigating to understand beneficial usage scenarios of (embedded) DSMLs in general.

7 Conclusion

We have pointed out the close relationship between the annotation capabilities in classical programming languages with the help of embedded DSLs and model transformation chains in a MDE process with the help of embedded DSMLs. However, we showed that, in particular for automated many-step model transformation chains, specific needs arise with regard to annotations in a MDE process.

Thus, it might happen that information defined at some point is not required in the next immediate transformation step, but in a later one. Also, a direct manipulation of the host models and their metamodel(s) is not an option.

Our approach utilizes embedded DSMLs, trace models and a megamodel. Based on our xWURST case study, concrete examples for each of these modelling artefacts have been given.

We implemented the proposed approach in a tool providing business performance related decision support. We experienced a high value of the approach as we are now able to annotate host models, in particular models conforming to the Business Process Modelling Notation and SAP proprietary models, with, for instance, business performance parameters. This enables us to utilize the Eclipse EMF based “MDPE Workbench” in combination with a SAP proprietary modelling tool and therefore to provide performance related decision support based on real-world business process models.

In future we anticipate gaining more experience with the proposed approach based on other business processes. Furthermore, it is planned to extend the current state of the MDPE process with functionality for guided business performance simulations. An open source version of the “MDPE Workbench” is planned as well.

Disclaimer

The information in this document is proprietary to the following MODELPLEX consortium members: SAP AG and TU-DRESDEN. The information in this document is provided “as is”, and no guarantee or warranty is given that the information is fit for any particular purpose. The above referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law. COPYRIGHT 2008 BY SAP RESEARCH TU-DRESDEN.

References

1. Hudak, P.: Building domain-specific embedded languages. *ACM Computing Surveys* 28(4), 196–196 (1996)
2. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: *OOPSLA 2004: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 365–383. ACM, New York (2004)
3. OMG: MetaObject Facility (MOF) specification version 2.0 (January 2006), <http://www.omg.org/cgi-bin/doc?formal/2006-01-01>
4. OMG: Mda guide version 1.0.1 (2003)
5. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. *Communication of the ACM* 15(12) (1972)
6. Fritzsche, M., Johannes, J.: Putting performance engineering into model-driven engineering: Model-driven performance engineering. In: Giese, H. (ed.) *MODELS 2008*. LNCS, vol. 5002, pp. 164–175. Springer, Heidelberg (2008)
7. Fritzsche, M., Gilani, W., Spence, I., Brown, T.J., Kilpatrick, P., Bashroush, R.: Towards performance related decision support for model driven engineering of enterprise soa applications, pp. 57–65. *IEEE Computer Society, Los Alamitos* (2008)
8. Fritzsche, M., Johannes, J., Zschaler, S., Zharebtsov, A., Terekhov, A.: Application of tracing techniques in model-driven performance engineering. In: *4th ECMDA Traceability Workshop (ECMDA-TW) Proceedings*, pp. 111–120 (2008)
9. OMG: Business Process Modeling Notation Specification, Final Adopted Specification (2006)
10. SAP AG: Review: SAPPHIRE 2008 - A new star is born in the BPM Galaxy
11. Fritzsche, M., Gilani, W., Fritzsche, C., Spence, I.T.A., Kilpatrick, P., Brown, 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)
12. Altenhofen, M., Hettel, T., Kusterer, S.: OCL support in an industrial environment. In: Kühne, T. (ed.) *MoDELS 2006*. LNCS, vol. 4364, pp. 169–178. Springer, Heidelberg (2007)
13. XJ Technologies: AnyLogic — multi-paradigm simulation software, <http://www.xjtek.com/anylogic/>
14. Knöpfel, A., Gröne, B., Tabelaing, P.: *Fundamental Modeling Concepts: Effective Communication of IT Systems*. John Wiley & Sons, Chichester (2006)
15. ATLAS Group: AMW Use Case - Model annotations in Java 1.4 (2007), <http://www.eclipse.org/gmt/amw/usecases/annotation>
16. Hillairet, G.: AMW Use Case - Metamodel Annotation for Ontology Design (2007), <http://www.eclipse.org/gmt/amw/usecases/oamusecase>
17. Vanhooft, B., Van Baelen, S., Joosen, W., Berbers, Y.: Traceability as input for model transformations. In: *ECMDA-FA 3th workshop on traceability* (2007)
18. Jouault, F.: Loosely Coupled Traceability for ATL. In: *ECMDA-FA workshop on traceability* (2005)
19. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On-demand merging of traceability links with models. In: *Proceedings: 2. ECMDA-FA workshop on traceability* (2006)
20. Barbero, Jouault, F., Bézivin, J.: Model driven management of complex systems: Implementing the macroscope’s vision. In: *15th ECBS 2008*, pp. 277–286. IEEE, Los Alamitos (2008)

21. The AMW Project Team: Atlas Model Weaver (June 2007),
<http://eclipse.org/gmt/amw/>
22. Sabetta, A., Petriu, D.C., Grassi, V., Mirandola, R.: Abstraction-raising transformation for generating analysis models. In: Bruel, J.-M. (ed.) *MoDELS 2005*. LNCS, vol. 3844, pp. 217–226. Springer, Heidelberg (2006)
23. ATLAS Group: ATLAS transformation language (June 2007),
<http://www.eclipse.org/m2m/at1/>
24. Bräuer, M., Lochmann, H.: An ontology for software models and its practical implications for semantic web reasoning. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) *ESWC 2008*. LNCS, vol. 5021, pp. 34–48. Springer, Heidelberg (2008)
25. Chen, K., Sztipanovits, J., Neema, S.: Toward a semantic anchoring infrastructure for domain-specific modeling languages. In: *EMSOF 2005: Proceedings of the 5th ACM international conference on Embedded software*, pp. 35–43. ACM, New York (2005)
26. Beživin, J., Pierantonio, A., Tratt, L. (eds.): *Intl. Workshop on Coordination of DSLs*, L'Aquila, Italy (September 2008)
27. Atkinson, C., Kühne, T., Henderson-Sellers, B.: Systematic stereotype usage. *Software and System Modeling* 2(3), 153–163 (2003)

Engineering a DSL for Software Traceability

Nikolaos Drivalos^{1,2}, Dimitrios S. Kolovos¹, Richard F. Paige¹,
and Kiran J. Fernandes²

¹ Department of Computer Science, University of York
{nikos,dkolovos,paige}@cs.york.ac.uk

² The York Management School, University of York
kf501@york.ac.uk

Abstract. The software artefacts at different levels of abstraction and at different stages of the development process are closely inter-related. For developers to stay in control of the development process, traceability information must be maintained. In this paper, we present the engineering of the Traceability Metamodelling Language (TML), a meta-modelling language dedicated to defining traceability metamodels. We present the abstract syntax of the language and its semantics, which are defined using a translational approach. Finally, we provide a case study that demonstrates the construction of a traceability metamodel that captures traceability information between two metamodels using TML.

1 Introduction

A Model Driven Engineering process typically involves a number of models that capture different - but potentially overlapping - views of the system under development. Moreover, with the advent of Domain Specific Languages, it is not uncommon for more than one modelling languages to be used in the context of the same MDE process. In this context, establishing and maintaining traceability links between related models elements across different models is of paramount importance in order to maintain control of the overall process and reason about design decisions across multiple viewpoints and levels of abstractions.

Following the discussion in [1], in [2] we argued that traceability information is a first-class MDE artefact and thus it should be maintained in the form of separate models that conform to semantically rich case-specific traceability metamodels and accompanying inter-model constraints. By applying this approach on a number of case studies - for each one of which we constructed a case-specific traceability metamodel and constraints - we have identified a number of reoccurring practices and patterns.

In this work, we propose the novel Traceability Metamodelling Language (TML) that elevates core traceability concepts and patterns into first-class meta-modelling artefacts. We present the abstract syntax of the language, and specify its semantics using a translational approach in which TML models are automatically transformed to respective ECore [3] metamodels and accompanying constraints.

The rest of the paper is organized as follows. Section 2 presents a brief review of the background on traceability information establishment and management and a discussion on the need for type safe, semantically rich traceability metamodels. In Section 3 we identify reoccurring patterns when constructing traceability metamodels and use them to motivate the construction of a meta-modelling language which elevates these patterns into first class meta-modelling artefacts. In Section 4 we present the abstract syntax and semantics of the Traceability Metamodeling Language (TML) and in Section 5 we briefly introduce how TML can be useful in traceability tool interoperability. In Section 6 we present a case study in which we use TML to specify a traceability metamodel. In Section 7 we provide an evaluation of the advantages and shortcomings of the proposed approach. In Section 8 we provide a discussion on related work and finally in Section 9 we conclude and discuss directions to further work on the subject.

2 Background and Motivation

The most common definition of traceability is the one provided in [4], according to which traceability is defined as:

The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example the degree to which the requirements and design of a given software component match.

For the purposes of investigating traceability in the context of MDE, a broader definition is required. This is due to the fact that the above definition focuses on horizontal traceability, which is usually met in the Requirements Engineering domain. A definition of traceability in the context of MDE should be broader and not focus on either horizontal or vertical traceability. Such a definition is provided in [5], where traceability is considered as any relationship that exists between artefacts of the software development life-cycle. In our work, we focus on relationships between elements of models or even between models, rather than on relationships between models and text or between text and text.

There are two main strategies in storing and maintaining traceability information [1]. In the first strategy, traceability links are embedded in the models they refer to, while in the second strategy traceability links are captured in a separate model dedicated to this purpose. As discussed in detail in [1,2] intra-model storage of traceability links is a human-friendly approach but progressively *pollutes* the models involved with information of secondary importance. By contrast, storing traceability links in the form of a separate model that conforms to a well-defined metamodel is not as human-friendly but demonstrates significant benefits in terms of consistency, quality and automation.

There are two alternatives for the traceability metamodel: use of a general-purpose traceability metamodel, or use of multiple case-specific traceability metamodels. In the following sections, we briefly discuss these approaches.

2.1 General Purpose Traceability Metamodel

In this approach, a generic metamodel that enables capturing relationships between any types of model elements is used. In this metamodel, a traceability link can connect any number of elements, of any type in any model. An example of an approach utilizing a generic traceability metamodel can be found in [6], where they develop a generic traceability metamodel, the Unified Traceability Scheme.

This approach favors simplicity and uniformity, since all the traceability metamodels, that are created conform to the same metamodel. As a result, tool interoperability can be achieved, since traceability information having the same format can be shared across heterogeneous traceability tools. However, due to its generic nature, this traceability metamodel can not express case specific structural constraints, such as the number or types of elements that can be connected in a traceability link and therefore a constraint language is needed to specify these structural constraints.

2.2 Case-Specific Traceability Metamodel

Under this approach, a case-specific traceability metamodel is defined for different traceability scenarios. Such a traceability metamodel is capable of capturing case-specific strongly typed traceability links with well-defined semantics that potentially include correctness constraints that extend beyond simple type conformance. The main advantage of this approach is its power to capture domain information and semantics, suited for different scenarios. The strongly typed nature of the metamodel in addition to its attached constraints restrict the establishment of illegitimate links. Additionally, as we discuss later in this paper, semantically rich traceability metamodels can be useful for producing additional MDE artefacts, such as Higher Order Transformations [7] for interoperating tools, which support general purpose traceability.

The main disadvantage of this approach is the complexity and effort associated with the creation of different case-specific traceability metamodels. Additionally, tools, which support different traceability metamodels can not exchange traceability information directly. Finally, every time a change occurs in the traceability requirements, this change must be “hard” coded into the case-specific metamodel. In the next sections of this paper, we demonstrate how the proposed approach addresses the aforementioned issues.

3 Reoccurring Patterns in Case-Specific Traceability Metamodels

Through experimentation with defining a number of case-specific traceability metamodels we have identified a set of reoccurring practices and patterns both in terms of the structure of the metamodel and of the additional constraints that guarantee the semantic integrity and completeness of the traceability models. In this section, we outline the identified patterns and demonstrate them through

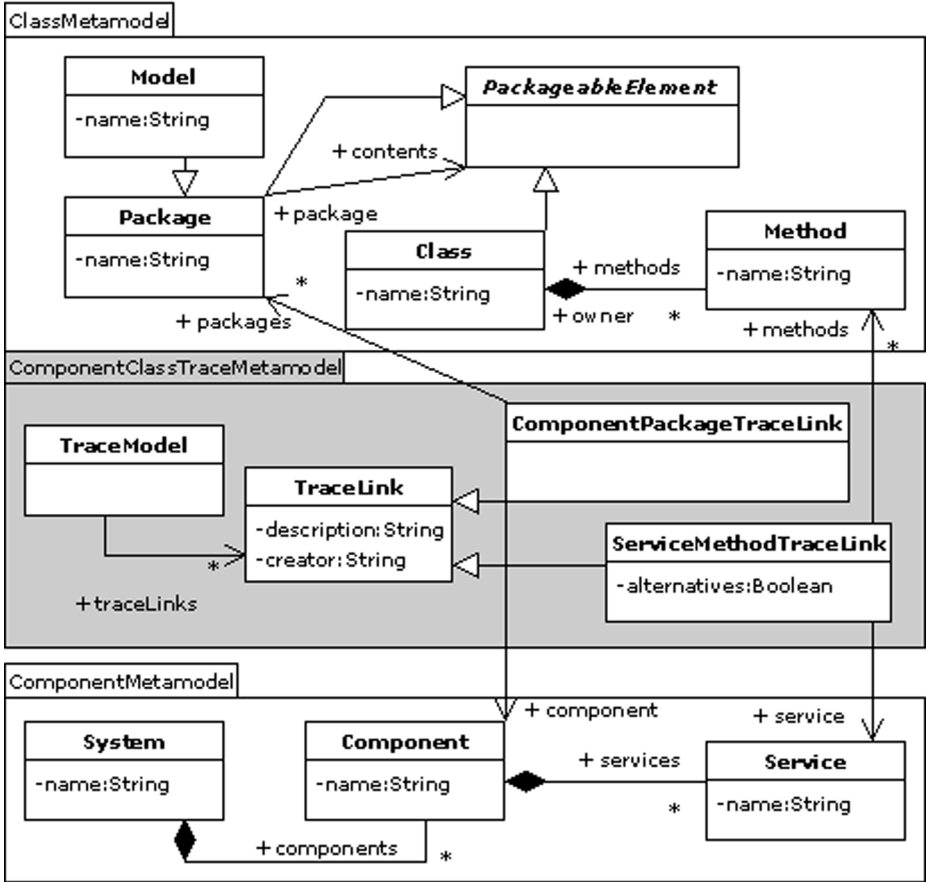


Fig. 1. The ComponentClassTraceMetamodel Traceability Metamodel

a simple – but representative – example. In our example, which is based on the example originally demonstrated in [2], we have hand-crafted a traceability metamodel (*ComponentClassTraceMetamodel*) for capturing traceability links between models that conform to two minimal *ClassMetamodel* and *ComponentMetamodel* metamodels. The three metamodels appear in Figure 1.

Through the process of defining traceability metamodels similar to the one presented in Figure 1 we have identified that each traceability metamodel contains a root class (typically named *TraceModel*) that acts as the root of a traceability model. Moreover, it contains a number of traceability link types (*ComponentPackageTraceLink* and *ServiceMethodTraceLink* in our example), all of which extend an abstract *TraceLink* abstract class for structuring purposes. Each traceability link contains a number of references of different types and cardinalities. For example, the *ComponentPackageTraceLink* metaclass contains the one-to-one *component* reference and the one-to-one *package* reference so that

it is able to trace one component to one package only. On the other hand, the *ServiceMethodTraceLink* metaclass contains the one-to-one *service* reference and the one-to-many *methods* reference so that it can trace one *service* to multiple *methods*.

Apart from the references that represent link ends (e.g. *packages*, *services*), each traceability link also typically stores some additional primitive information. This information either applies to all traceability links (e.g. the *description* and *creator* attributes in meta-class *TraceLink*) or to some particular types of traceability links only (e.g. the *alternatives* attribute in the *ServiceMethodTraceLink* meta-class which shows if the service depends on all of the methods (logical and) or only on one of them (logical or)).

Moreover, as discussed in [2], establishing a traceability metamodel extends beyond constructing the abstract syntax and also involves specifying validity constraints. Here we discuss recurring examples of such constraints:

ForAll: It is often required that at least one traceability link of a particular kind exist for all instances of a type. In our example, we require that at least one service-method traceability link exists for each service in the *Component* model.

Unique: Another common constraint is when it is required that at most one traceability link of a particular kind exists for each instance of a given type. In our example it is required that each *component* can link to at most one *package* to avoid fragmentation.

Optional: Additional information captured in the form of primitive attributes (e.g. *description*) is often neglected by end-users, thus resulting in poorly documented traceability models. For this purpose, it is typical that additional constraints are specified that ensure that the values of the attributes in the trace models are not empty. Apart from these reoccurring patterns, there are other case-specific constraints that cannot be generalized. For instance, in our example such a constraint is that *the methods a service traces to belong to classes contained in a package which is traced to the component that provides the service*.

The existence of reoccurring patterns hints that a higher-level of abstraction is potentially beneficial for defining traceability metamodels. Under this rationale, in the next section, we attempt to promote the identified patterns into first-class artefacts in a metamodeling language dedicated to the construction of traceability metamodels.

4 The Traceability Metamodeling Language (TML)

The purpose of the Traceability Metamodeling Language (TML) is to enable the construction and maintenance of traceability metamodels and accompanying constraints with reduced effort by encoding constructs and relationships that are often encountered when constructing traceability metamodels into first-class metamodeling artefacts.

In three-level metamodeling architectures such as MOF and EMF, there is only one metamodeling language (language for constructing metamodels) and that is MOF and ECore respectively. Therefore, defining a metamodeling language in

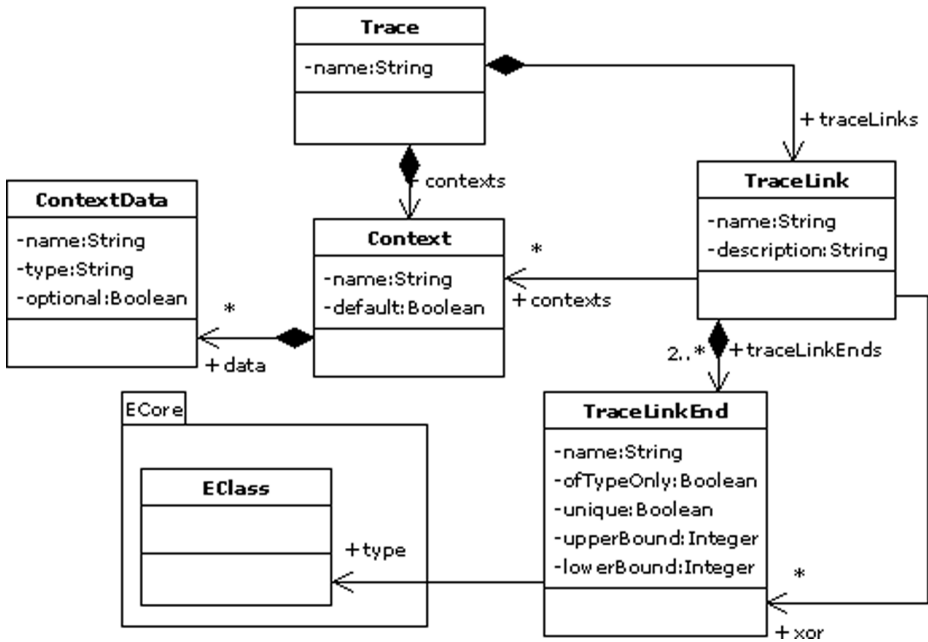


Fig. 2. The Traceability Metamodelling Language

the context of such an architecture appears to be a contradiction. Therefore at this point we should clarify that strictly speaking, TML is not a metamodelling language but a modelling language as its metamodel is expressed using ECore. However, TML models (instances of the TML metamodel) are eventually transformed into ECore metamodels - which conceptually renders TML a metamodelling language. This is further clarified in the sequel.

4.1 Abstract Syntax

As discussed above, the abstract syntax of TML has been defined using ECore and is presented in Figure 2. In this section, the concepts of the TML metamodel are discussed in detail.

Trace: Acts as the root of a TML model. A Trace defines a name and can contain a number of TraceLinks and Contexts.

TraceLink: Represents a traceability link between a number of elements. It defines a name and contains a number of TraceLinkEnds. It also associates to a number of contexts through which it can capture custom information.

TraceLinkEnd: Represents an end of a traceability link. It defines a name and the type of elements it can link to. Through the *ofTypeOnly* attribute it also defines if it can link to elements of the particular type only or if it can also link to instances of all the subtypes of the type. The unique attribute defines if more than

one traceability link linking to the same model element of this type are legitimate. The *forAll* attribute specifies if it is mandatory for all elements of that type to exist a traceability link of this kind. The *lowerBound* and *upperBound* attributes specify how many elements of the specified type can be linked to the same end.

Context: The role of the context metaclass is to enable traceability metamodel designers to attach custom information to traceability links. Each context defines a number of *ContextData* and can be specified as *default*, which means that it is implicitly associated to all *TraceLinks* (as opposed to the explicit *data* reference of class *TraceLink*).

ContextData: Captures additional information of primitive type (currently the String, Boolean and Integer types are supported) about a *TraceLink*. It defines a name, a type and whether it is optional or not.

4.2 Semantics

The semantics of TML are specified in a translational manner [8] using two formal and executable transformations. These transformations transform a TML model into an ECore metamodel and a set of constraints expressed using the Epsilon Validation Language (EVL) which is an extension of OCL [9].

Transforming to an ECore metamodel. As discussed above, a TML model lives in the M1 level of the three-tiered metamodeling architecture, and thus it cannot be instantiated as-is. Therefore, in order to instantiate it we need to transform it into an M2 level model (proper metamodel). Listing 1.1 demonstrates such a transformation using the Epsilon Transformation Language (ETL) [10], although in principle any model-to-model transformation language (e.g. ATL[11], QVT[12], Kermeta [13]) could have been used for this purpose. In this section we go through the transformation and discuss its functionality.

Listing 1.1. The TML2ECore Transformation expressed in ETL

```

1  pre {
2    var traceModel := new ECore!EClass;
3    traceModel.name := 'TraceModel';
4
5    var traceLink := new ECore!EClass;
6    traceLink.name := 'TraceLink';
7    traceLink."abstract" := true;
8
9    var modelLinksReference := new ECore!EReference;
10   modelLinksReference.name := 'links';
11   modelLinksReference.eType := traceLink;
12   modelLinksReference.containment := true;
13   modelLinksReference.upperBound := -1;
14   traceModel.eStructuralFeatures.add(modelLinksReference);
15 }
16
17 rule Trace2Package
```



```

18   transform s : Trace!Trace to t : ECore!EPackage {
19       t.name := s.name;
20       t.nsUri := t.name;
21       t.eClassifiers.add(traceModel);
22       t.eClassifiers.add(traceLink);
23       t.eClassifiers.addAll(s.links.equivalent());
24       t.eClassifiers.addAll(s.contexts.equivalent());
25   }
26
27   rule TraceLink2EClass
28       transform s : Trace!TraceLink to t : ECore!EClass {
29           t.name := s.name;
30           t.eSuperTypes.add(traceLink);
31           t.eStructuralFeatures.addAll(s.ends.equivalent());
32           t.eSuperTypes.addAll(Trace!Context.all.select(c|c.default).
33               equivalent());
34           t.eSuperTypes.addAll(s.contexts.equivalent());
35       }
36
37   rule TraceLinkEnd2EReference
38       transform s : Trace!TraceLinkEnd to t : ECore!EReference {
39           t.name := s.name;
40           t.eType := s.type;
41           t.containment := false;
42           t.lowerBound := s.lowerBound;
43           t.upperBound := s.upperBound;
44       }
45
46   rule Context2EClass
47       transform s : Trace!Context to t : ECore!EClass {
48           t.name := s.name;
49           t."abstract" := true;
50           t.eStructuralFeatures.addAll(s.data.equivalent());
51       }
52
53   rule ContextData2EAttribute
54       transform s : Trace!ContextData to t : ECore!EAttribute {
55           t.name := s.name;
56           t.eType := s.type.literal.createEType();
57       }
58
59   @cached
60   operation String createEType() {
61       var type := new ECore!EDataType;
62       type.name := self;
63       type.instanceClassName := 'java.lang.' + self;
64       Trace!Trace.all.first().equivalent().eClassifiers.add(type);
65       return type;
66   }

```

The *pre* part of the transformation in lines 2–14 creates the basic classes in all traceability metamodels, namely *TraceModel* and *TraceLink* which act as the container and the root-type for links respectively. It also specifies that a *TraceModel* contains zero or more *links*.

Rule *Trace2EPackage* creates a new *EPackage* from the TML *Trace*, adds the default *TraceModel* and *TraceLink* metaclasses created in the *pre* section of the transformation, sets its name and namespace URI and adds the *EClasses* generated from the *traceLinks* and *contexts* of the *Trace* to the list of *classifiers* of the *EPackage*.

Rule *TraceLink2EClass* creates a new *EClass* for each *TraceLink* and line 27 sets its name to be the same as the *TraceLink* and then implicitly calls the *TraceLinkEnd2EReference* that creates a new *EReference* for each *end* of the *TraceLink* with respective cardinality (upper and lower bounds).

Rules *Context2EClass* and *ContextData2EAttribute* transform each context and its respective data into an *EClass* and a set of *EAttributes* in lines 46 and 53 respectively.

Transforming to EVL Constraints. To generate the constraints that complement the ECore metamodel, we have employed a template-based model-to-text transformational approach using EGL [14] as the template language. Again, any model-to-text language such as MOFScript [15], JET [16] or XPand [17] could have been used instead. Listing 1.2 demonstrates an excerpt from the EGL template that transforms a TML metamodel into a set of constraints¹.

In this excerpt, for all *TraceLinkEnds* that have their *forAll* attribute set to true (line 2), a constraint is generated in line 4. In line 6, the guard of the constraint is generated by calling the *computeGuard()* operation defined in line 24 which returns an appropriate guard according to the value of the *ofTypeOnly* attribute of the *TraceLinkEnd*.

In lines 10 and 12, two alternative constraint bodies (*check*) are generated depending on the cardinality of the processed end. In line 16 the *message* part of the constraint - which is evaluated if the constraint is not satisfied for a particular model element - is specified.

Listing 1.2. The TML2EVL model to text transformation expressed in EGL

```

1 [%for (link in Trace!TraceLink.allInstances) { %]
2   [%for (end in link.ends.select(e|e.forAll = true)) { %]
3     context [%=end.type.name%] {
4       constraint ForEach[%=end.name.firstToUpperCase()%]{
5
6         [%=end.computeGuard()%]
7
8         [%if (end.isMany()) {%]
9           check : [%=link.name%].all.exists(e|e.[%=end.name%].
10             includes(self))

```

¹ Due to space restrictions, the complete M2T template and the M2M transformation of Listing 1.1 are available online in <http://www.cs.york.ac.uk/~nikos/research/TML.zip>

```

11     [%] else [%]
12     check : [%=link.name%].all.exists(e|e.[%=end.name%]=self)
13     [%]
14
15
16     message: 'No [%=link.name%] found for [%=end.name%]'+self
17
18     }
19     }
20     [%]
21 [%]
22
23 [%
24 operation Trace!TraceLinkEnd computeGuard() : String {
25     if (self.ofTypeOnly) {
26         return 'guard : self.isTypeOf(' + self.type.name + ')';
27     }
28     return '';
29 }
30
31 operation Trace!TraceLinkEnd isMany() : Boolean {
32     return self.upperBound > 1 or self.upperBound = -1;
33 }
34
35 %]

```

Additional constraints are generated based on the value of the *unique* attribute, to ensure that only one traceability link is allowed for each instance of a given type. Similarly, yet more constraints are generated and based on the value of the *optional* attribute of *ContextData* to ensure the non-emptiness of mandatory contextual information.

In this section we have presented the abstract syntax of TML and defined its semantics using a translational approach. In section 6 section we present a concrete case study to demonstrate the practicality of the approach and - hopefully - clear any confusion that has been generated so far due to the cross-level and cross-language nature of the transformations presented here.

5 Interoperability between Heterogeneous Traceability Tools

In section 2.2, we identified the tool interoperability issue as one disadvantage related to case-specific traceability metamodels. That is, tools which support different traceability metamodels, can not exchange traceability models directly. In this section, we will briefly describe how the use of model engineering principles and TML can address this issue.

In general, to enable interoperation between traceability tools, the artefacts created with one tool should be somehow accessible from other tools. In the context of MDE, the information described by a traceability tool is considered

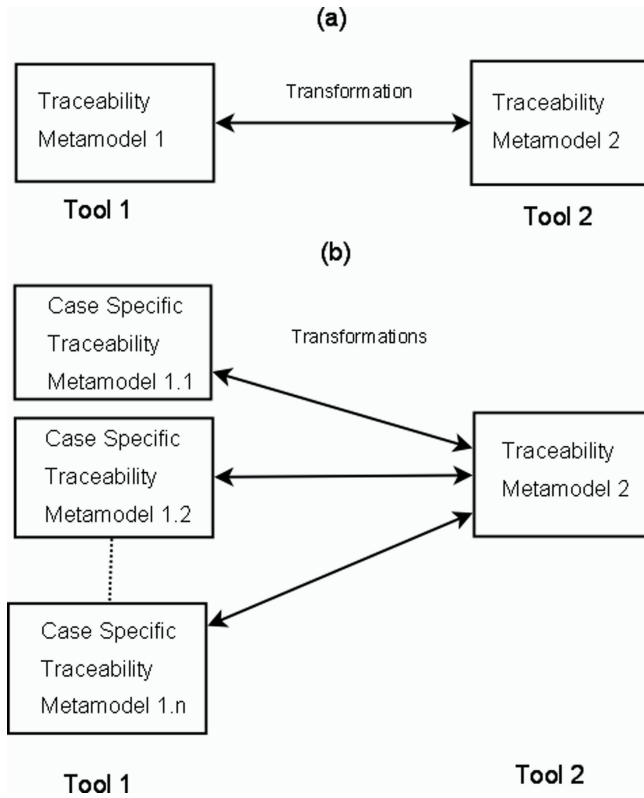


Fig. 3. Overview of the model engineering approach to interoperability

to be a model that conforms to a given metamodel. Since it is possible for the various traceability tools to support different traceability metamodels, the differences at the syntactical and semantic level of the created artefacts must be reconciled.

In the case where we need to share traceability information between tools which support a general purpose traceability metamodel, interoperability is achieved by specifying a transformation from the traceability metamodel of one tool to the metamodel of the other tool (figure 3(a)). This transformational approach is very common (eg. [18,19]). In this case only one transformation is sufficient since general purpose metamodels are used by both tools.

The second case is that a tool supports case-specific traceability metamodels. In order to achieve interoperability between tools in this case, a transformation has to be defined for each case-specific metamodel of one tool and the traceability metamodel of the other one. In the case that the target tool supports case-specific traceability metamodels as well, we treat each of those metamodels as a different tool. This approach is illustrated in figure 3(b). The added complexity and required effort associated with the interoperability of tools which support case-specific traceability metamodels is significant.

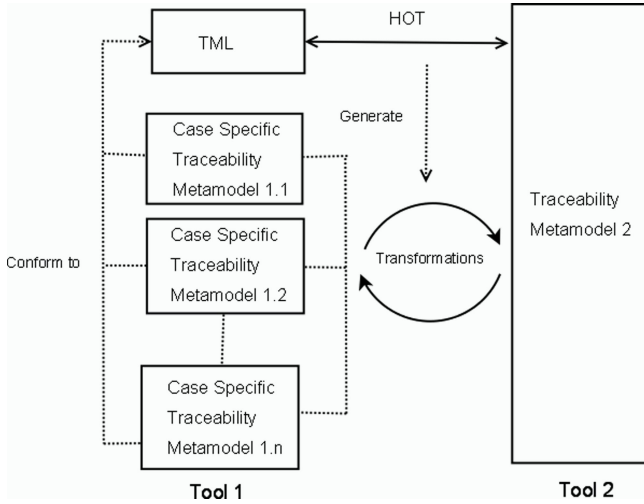


Fig. 4. Using TML for tool interoperability

By using the TML as a traceability metamodeling language, we can address the aforementioned tool interoperability issue. Since TML has rich semantics it can be used to generate the necessary transformations between each case-specific metamodel of one tool and the metamodel of the other tool in a more systematic and automated manner. This is achieved by specifying a Higher Order Transformation (HOT) [7], i.e. a transformation which will generate automatically the necessary transformations. This transformation will take as source the TML metamodel and the metamodel of the respective traceability tool. This concept is illustrated in figure 4.

The sole purpose of this brief discussion on how to use TML to address the interoperability issue of case-specific traceability metamodels was purely for demonstrating the potential of a dedicated metamodeling language for traceability. Further discussion about this technique or other possible uses of TML is considered to be out of the scope of this paper.

6 Case Study

Following [2], in Section 3 we have developed a traceability metamodel in ECore (*ComponentClassTraceMetamodel*) for capturing traceability links between models that conform to two minimal *ClassMetamodel* and *ComponentMetamodel* metamodels (Figure 1). In this section, we will attempt to reproduce the hand-crafted traceability metamodel and a subset of the associated constraints using TML instead. Our aim is to capture traceability links between instances of *Package* from the *ClassMetamodel* and *Component* from the *ComponentMetamodel*, and links between instances of *Method* from the *ClassMetamodel* and *Service* from the *ComponentMetamodel*. Furthermore, we will produce a set of related constraints directly from the TML traceability metamodel.

6.1 Defining the ComponentClass Metamodel in TML

The first step of specifying the TML metamodel illustrated in Figure 5 is to define the root element, i.e. the *Trace* element. In our example we call this element *ComponentClassTraceMetamodel*. Then, the two traceability links (*ComponentPackage* and *ServiceMethod*) are created. Each traceability link is associated with link ends. In the case of the *ComponentPackage* link, there are two link ends associated with it, the *package* link end of type *Package* from the *ClassMetamodel* and the *component* link end of type *Component* from the *ComponentMetamodel*.

Each link end has a set of attributes. The *package* link end has the attribute *forAll* set to false, which means that it is not mandatory for all elements of type *Package* to be involved in a traceability link. In addition, the upper and the lower bound for the *package* link end are set to 1, limiting its multiplicity to 1. The attribute *ofTypeOnly* is boolean, and it is set to false. This means, that this link end can be only of type *Package* and it can not be of type *Model*, which extends the class *Package* in the *ClassMetamodel*. Finally, the attribute *unique*, which ensures that at most one traceability link of a particular kind exists for each instance of a given type, is set to true. This will be later interpreted as a constraint that each *component* can link to at most one *package*.

The other link end of the *ComponentPackage* link is the *component* link end of type *Component* from the *ComponentMetamodel*. The *forAll* attribute is set to true, meaning that all components must link to a package. Additionally, the upper and the lower bound for the *component* link end are set to 1, while the

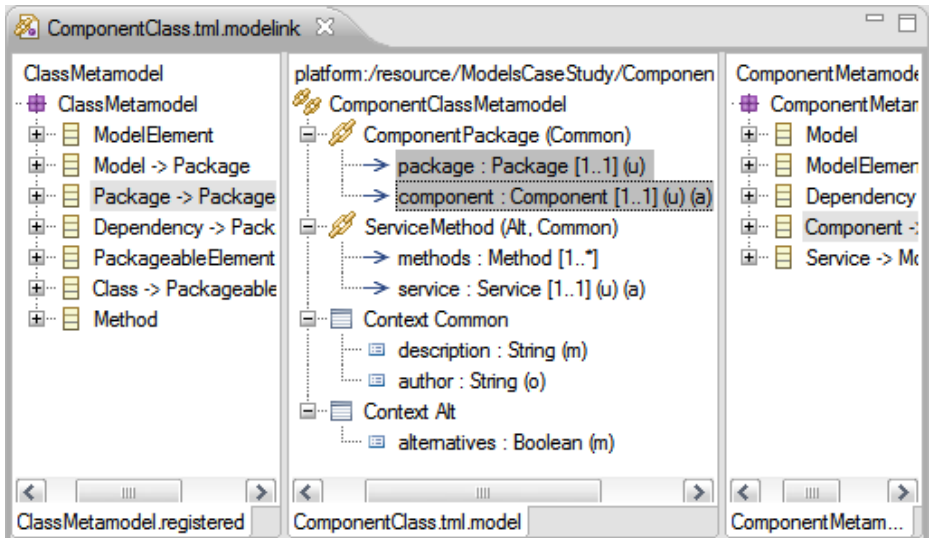


Fig. 5. The ComponentClassTraceMetamodel Traceability Metamodel created with TML

ofTypeOnly attribute's value is false. Finally, the *unique* attribute is set to true which means that each *component* can only be traced to one *package*.

Similarly, the *ServiceMethod* link is associated with two link ends, the *methods* link end of type *Method* from the *ClassMetamodel* and the *service* link end of type *Service* from the *ComponentMetamodel*. These link ends have the same set of attributes. The value of the *forAll* attribute is false for the *methods* link end, while it is true for the *service* link end. Furthermore, the multiplicity for the *methods* link end is set to be one to many, while the multiplicity for the *service* link end is set to be one to one. For both of these two link ends, the value of the attribute *ofTypeOnly* is false. Finally, the *methods* link end's *unique* attribute is set to false, whereas the same attribute for the *service* link end is set to true.

Apart from the links and their associated link ends, the TML metamodel specifies a number of contexts, which can be associated to one or more of the traceability link types. The first context specified in our example is the *Common* context, whose *default* attribute is set to true. By setting this attribute to true, we associate this context explicitly to both of the aforementioned links of our model. Each context of TML has a number of *ContextData*. In our example, the *Common* context has two, namely *description* and *author*. The former provides a short description for each traceability link, while the latter provides the creator of each link. Each *ContextData* has three attributes associated with it, i.e. *name*, *type* and *optional*. The *description ContextData* is of type String and its *optional* attribute is set to false. This means that it is mandatory that every link in our model has a description. The *author ContextData* is of type String as well and its *optional* attribute is set to true, meaning that it is not mandatory to specify an author for each traceability link in our exemplar model.

The second context of our example is the *Alternatives* context, which is not attached to all of the trace links in the model, but only to the *ServiceMethod* traceability link. Hence, the *default* attribute of this context is set to false. Its *ContextData* is named *alternatives*, and shows if a service depends on all of the methods (logical and) or only on one of them (logical or). In our example, this is of type boolean and it is mandatory.

6.2 Generating the ECore-Based Metamodels and the EVL Constraints

After creating a TML model, we can use the M2M and M2T transformations presented in Section 4 to obtain a traceability metamodel in ECore and a set of associated constraints expressed in our case in EVL. The generated, case-specific traceability metamodel in ECore is semantically equivalent to the hand-crafted one presented in Figure 1. In addition, the specified attributes for the various TML elements, are used to generate constraints in EVL. For example, the *forAll* attribute of the *component* link end, which is set to true in our example, produces the EVL constraint of Listing 1.3 while the *optional* attribute of the *description* context data which is set to false produces the constraint of Listing 1.4. Due to space restrictions, it is not possible to present all the generated constraints here;

the complete list of constraints as well as the produced metamodel can be found online.²

Listing 1.3. The EVL constraint generated by the *forAll* attribute

```

1 context Component {
2   constraint OneForEachComponent {
3     check : ComponentPackage.all.exists(e|e.component = self)
4     message: 'No links of type ComponentPackage found for component'
5     + self
6   }
7 }
```

Listing 1.4. The EVL constraint generated by the *optional* attribute

```

1 context Common {
2   constraint NotOptionalDescription {
3     check : self.description.isDefined()
4     and self.description.trim() <> ''
5     message : 'No value specified for attribute description'
6   }
7 }
```

7 Evaluation

We have evolved and refined the syntax and semantics of TML over a number of diverse case studies. In its current form, TML is capable of capturing complex traceability metamodels and to our view the language is well-balanced between expressiveness and domain specificity. A complex example of using TML for specifying a traceability metamodel for the i* and KAOS metamodels from the requirements engineering domain is available online³.

A downside of the proposed approach is that metamodels designed using TML adopt an explicit id-based linking approach [20]. As a consequence, if an element involved in a trace link is deleted, there is no information which can assist a user (or a tool) to identify alternative traceability targets. To address this problem we are working on extending TML with constructs that can capture and maintain redundant information for traced elements, which can then be used to semi-automatically restore broken traceability links.

8 Related Work

While there is a significant amount of work in the field of model-to-model traceability, during our review we have not encountered approaches similar to the one presented in this work.

² www.cs.york.ac.uk/~nikos/research/ModelsCaseStudy.zip

³ <http://www.cs.york.ac.uk/~nikos/research/ISarKAOS.zip>

Of the closest relation to our work is AMW, the Atlas Model Weaver [21], which is a tool that can capture and store links between models of diverse EMF-based metamodels. In AMW, links between model elements are stored in a model, which is called *weaving model*. AMW uses a core weaving metamodel which users can extend in order to specify new types of links. However, AMW weaving metamodels are not strongly typed (i.e. a link cannot specify that it can only connect packages to components in terms of the example presented in Section 6).

9 Conclusions and Further Work

In this paper we have argued over the benefits of type safe and semantically rich traceability metamodels and presented TML, a language for constructing and maintaining rigorous traceability metamodels at a high level of abstraction. We have demonstrated the practicality of our approach by applying it on a small but representative case study.

In section 2, we have identified concerns associated with case-specific traceability metamodels, and then shown how TML contributes to addressing them. The first concern is the associated complexity with constructing such metamodels. In this paper, we argue that this complexity can be managed to an extent by using a dedicated metamodeling language. This is achieved by encoding constructs and relationships that are often encountered during the construction of traceability metamodels into first-class language constructs. The second issue is related to tool interoperability. In section 5 we have briefly described how TML, augmented with higher-order transformations, can be used to address this issue.

We are currently exploring how TML can be used to automate traceability maintenance and in particular, how broken traceability links can be semi-automatically recovered by extending TML with support for capturing and managing redundant information.

This work has also raised a broader issue, which has to the best of our knowledge not been previously reported; that of Domain Specific Metamodeling Languages (DSM2Ls). Similarly to the way a DSL is beneficial for designing a particular kind of models, a DSM2L can be more suitable than a general-purpose metamodeling language (such as MOF or ECore) for designing a particular kind of metamodels (e.g. as TML is for designing traceability metamodels). In the near future we plan to perform further exploration in order to identify similar families of metamodels for which a DSM2L can be more beneficial compared to a general-purpose metamodeling language.

References

1. Kolovos, S.D., Paige, R.F., Polack, F.A.C.: On-Demand Merging of Traceability Links with Models. In: Proc. 2nd EC-MDA Workshop on Traceability, Bilbao, Spain (July 2006)
2. Drivalos, N., Paige, R.F., Fernandes, K.J., Kolovos, D.S.: Towards Rigorously Defined Model-to-Model Traceability. In: Proc. 4th Traceability Workshop, ECMDA, Berlin, Germany (June 2008)

3. Eclipse.org. Eclipse Modelling Framework, <http://www.eclipse.org/emf>
4. IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology. IEEE, New York
5. Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafini, Y.: Model Traceability. IBM Systems Journal (2006)
6. Limon, A.E., Garbajosa, J.: The Need for a Unifying Traceability Scheme. In: Proc. Traceability Workshop, European Conference in Model Driven Architecture (EC-MDA), pp. 47–55 (2005)
7. Jouault, F., Bezivin, J.: Using ATL for Checking Models. In: Proc. International Workshop on Graph and Model Transformation (GraMoT), Tallinn, Estonia (September 2005)
8. Kleppe, A.: A Language Description is More than a Metamodel. In: Proc. 4th International Workshop on Software Language Engineering, Nashville, USA (October 2007)
9. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In: Proc. Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis (2007)
10. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The epsilon transformation language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 46–60. Springer, Heidelberg (2008)
11. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
12. Object Management Group. MOF QVT Final Adopted Specification, <http://www.omg.org/cgi-bin/doc?ptc/05-11-01.pdf>
13. Chauvel, F., Fleurey, F.: Kermeta Language Overview, <http://www.kermeta.org>
14. Rose, L.M.: The Epsilon Generation Language (EGL). MEng. Thesis, Department of Computer Science, The University of York (2008)
15. Oldevik, J.: MOFScript User Guide, <http://www.eclipse.org/gmt/mofscript/doc/\MOFScript-User-Guide.pdf>
16. Java Emitter Templates (JET), <http://www.eclipse.org/modeling/m2t/>
17. Sven Efftinge. XPand Language Reference, http://www.eclipse.org/gmt/oaw/doc/4.1/r20_xPandReference.pdf
18. Maedche, A., Motik, B., Silva, N., Volz, R.: MAFRA – A mapping framework for distributed ontologies. In: Gómez-Pérez, A., Benjamins, V.R. (eds.) EKAW 2002. LNCS, vol. 2473, p. 235. Springer, Heidelberg (2002)
19. Bézivin, J., Brunelière, H., Jouault, F., Kurtev, I.: Model engineering support for tool interoperability. In: WISME 2005 - 4th Workshop in Software Model Engineering (2005)
20. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Detecting and Repairing Inconsistencies Across Heterogeneous Models. In: Proc. 1st IEEE International Conference on Software Testing, Verification and Validation, Lillehammer, Norway (April 2008)
21. Fabro, M.D.D., Bezivin, J., Jouault, F., Breton, E., Gueltas, G.: AMW: A Generic Model Weaver. In: Proceedings of IDM 2005 (2005)

Towards an Incremental Update Approach for Concrete Textual Syntaxes for UUID-Based Model Repositories

Thomas Goldschmidt

FZI Research Center for Information Technology
Karlsruhe, Germany
goldschmidt@fzi.de

Abstract. Textual concrete syntaxes for models are beneficial for many reasons. They foster usability and productivity because of their fast editing style, their usage of error markers, autocompletion and quick fixes. Several frameworks and tools from different communities for creating concrete textual syntaxes for models emerged during recent years. However, these approaches failed to provide a solution in general. Open issues are incremental parsing and model updating as well as partial and federated views. On the other hand incremental parsing and the handling of abstract syntaxes as leading entities has been solved within the compiler construction communities many years ago. In this short paper we envision an approach for the mapping of concrete textual syntaxes that makes use of the incremental parsing techniques from the compiler construction world. Thus, we circumvent problems that occur when dealing with concrete textual syntaxes in a UUID based environment.

1 Introduction

An important part of a modelling language apart from the metamodel itself is its concrete syntax. Today's languages mostly have either a graphical or a textual concrete syntax. In several use cases a concrete textual syntax (CTS) is more appropriate to edit models than a graphical syntax. For example, this applies to mathematical, expression-like languages such as query, or constraint languages (e.g. Object Constraint Language (OCL)[1]). Another example where textual syntaxes are preferred over graphical ones are model transformation languages such as QVT. Even in graphical modelling there are parts that can only be expressed and displayed textually in a convenient way, e.g., an operation signature in UML.

Several approaches that provide solutions to automatically generate parsers, emitters (also called unparsers) and editors for metamodels from a grammar like description appeared during recent years. Most of these approaches discard and re-create all model elements upon re-parsing of the textual syntax. Depending on the identification mechanism of the underlying model repository this is a huge problem. There are two different possibilities how model repositories can handle links between model elements: either by defining designated key attributes for each model element (as, e.g., possible in EM-F/ECore) or by assigning each element a Universally Unique Identifier (UUID) that remains stable across the lifetime of the element (e.g. the MOFID in MOF 1.4) [2,3]. Very important issues arise when trying to put a parser-based approach on top of a repository

that uses UUIDs to identify model elements. In large scale environments with a high number of model partitions and numerous connections between those partitions, such repositories become very important. In distributed development where developers of one artefact do not always know all referrers from other model partitions to a specific model element it is crucial that elements have stable IDs. Further advantages of the UUID-based approach can be found in [2].

On the other hand, approaches for incremental parsing exist since many years (e.g., [4]. Even more sophisticated systems which can be used to generate whole development environments including language specific editors have been developed in the compiler construction community many years ago [5][6][7]. However, such techniques were not adopted by any of the CTS frameworks under evaluation.

In this paper we envision a CTS approach that combines and enhances techniques from both worlds. By using incremental parsing techniques and heuristic analysis of the changes that occurred since the last parsing we try to retain as many model elements as possible from the previous version of the model.

2 Requirements

Having a Universally Unique Identifiers (UUIDs) based model repository poses some additional requirements to the implementation of a CTS approach. UUIDs for model elements and therefore also links between model elements that are based on these UUIDs links unfold several advantages in a large-scale setup [3]. Especially for distributed and parallel model development this identification mechanism is important. Imagine in a key attribute based repository one user changes the key attribute of an element that is referenced from several other elements that he does not necessarily have under control or perhaps does not even know about. All references to this element will break. This problem does not occur with UUID based identification. However, UUID based identification comes at a price that currently not all repository and tool implementations are willing to pay. Caution is therefore required in selecting the right infrastructure components for an enterprise modelling setup. There are several issues that have to be tackled when handling UUID based elements, though especially when a CTS should be developed that does not store these IDs explicitly in the textual representation as well.

2.1 Lifecycle

While its UUID remains stable across the life time of a model element, some repositories allow the key attributes to change their values. For example, the Web Tools Platform (WTP) built in Eclipse with EMF uses names for identifying model elements. Element references can break if elements change their name and referring elements are not covered by the refactoring. UUID-based references are not affected by such changes and from this perspective work better in a large-scale environment where owners of an artefact do not always know all of the artefact's users or referrers. Beyond that, UUIDs can get "lost" if elements are accidentally deleted. It then depends on the tools and the capabilities of the repository how this case gets handled and what this means to references pointing to the UUID which now has disappeared.

In textual syntaxes identification through UUIDs becomes problematic for several reasons:

- **Storage mechanism:** If a model artefact is stored using the concrete syntax only, there needs to be the possibility to store these IDs somewhere in the text. However, during development a developer should not see these IDs as they contradict the crisp textual view and make it confusing. On the other hand, if the artefact is solely stored as model, other problems concerning the update from text to model arise (see below).
- **Model updates:** Updating existing models is an inherent problem of textual notations. In graphical or forms-based modelling only a comparably small set of changes may occur that can easily be wrapped into command structures that are executed in a transactional way, transforming the model from one consistent state to another. In a textual editor this is very difficult, especially when IDs are not present in the textual representation. A small change, e.g., adding an opening bracket in the text may alter the whole structure of the text, making it difficult to identify which elements in the model are meant to be kept and which are not.
- **Creation and deletion of model elements:** In a graphical editor there are explicit commands to create and delete model elements. Within a textual editor this is difficult mostly because the creation or deletion of model elements is done implicitly. For example, if the name of an element is changed in the textual syntax it may either mean that the old element should be deleted and a new one should be created, or a simple rename of the existing model element may have been intended.

A solution that lets transformations produce stable UUIDs for new model elements was proposed in [3]. For example, such an approach may use the ID of the source element and some transformation ID to compute the ID of the target element. However, if text is used as source there is no stable ID for a source element, just properties derived directly from the textual representation, such as a name attribute. Hence, no stable ID for a target element can be computed and the only way to keep the identity is to rely on incremental updates of the model.

2.2 Partial and Combined Views

One advantage of graphical modelling is that it is easily possible to define partial views on models. This means that it is possible to create diagrams that highlight only a specific aspect of the model while hiding other parts. For example, in UML one diagram may be used to display an inheritance hierarchy of classes only while another diagram is used to show the interactions between these classes. Defining models with a CTS should also include the possibility to do this. However, this implies that there are two different modes for deleting elements. One, which deletes only the text and another which deletes the model element from the text *and* the model. Using only standard text editing techniques (typing characters and using backspace or delete to remove them) there is no possibility to distinguish between both commands.

3 Related Work

During recent years many approaches for defining a concrete textual syntaxes for meta-models emerged from the model-driven community. Most of these approaches ([8],[9],[10],[11],[12],[13],[14] are based on the Ecore meta-meta-model [15]. An assumption made by most of these approaches is that it is possible to re-create model elements from the concrete textual syntax without losing its identity. A detailed analysis of the features of these approaches can be found in [16]. All these approaches have in common that they don't consider incremental updates from the textual to the abstract syntax. Therefore it is not possible to use these approaches in an UUID based environment without losing references to the created elements each time the textual syntax is changed.

The idea of having a tighter integration of concrete and abstract syntax was already thoroughly researched in the compiler construction community several years ago. For example, the IPSEN approach [5] introduced a tightly integrated software development environment, including incremental update and storage of the abstract syntax elements as leading entities. Also the generation of the environment including editors featuring syntax directed autocompletion and error checking could already be generated out of a attributed grammar. Other approaches that are based on this idea are the Cornell Language Synthesizer [6] and the PSG - Programming System Generator [7]. The incremental update processes of these approaches were mainly focused on reducing the effective compilation time of the programs. Furthermore, the way of editing within these approaches was either completely restricted to a syntax directed editing or completely decoupled from the abstract syntax without guidance during editing.

4 Envisioned Approach

The focus of our work is not to invent a new CTS mapping approach but rather concentrate on the problematic updates that occur with existing approaches in UUID-based repositories. In theory the approach presented here is, despite of some assumptions that have to be fulfilled, applicable to different existing CTS approaches.

4.1 Model-View-Controller Pattern

Many of the aforementioned problems may be solved by employing a Model-View-Controller (MVC) based update mechanism. Here all updates that are done in the text are directly reflected in the model. Thus, keeping the model in a consistent state at any time and also ensuring that model and text are in sync. However, this approach does not allow the certain degree of freedom which developers nowadays are used to. Intelligent IDEs such as Eclipse offer a lot of functionality that is based on the assumption that the edited text may be inconsistent to a certain degree. For example, typing a method call before a method exists and then using the quick fix functionality to automatically create the corresponding method is such a feature. A discussion of the advantages and disadvantages of the MVC and the background parsing approach can be found in [13]. Also the compiler construction literature there are similar approaches that are called *Syntax Directed Programming Environments*[6]. In these approaches commands are derived

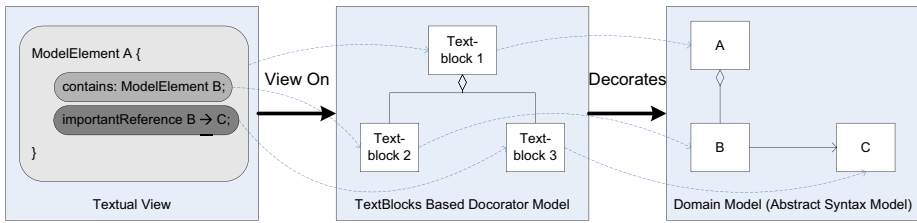


Fig. 1. Decorator Model - Overview

from the (attributed) grammar that can be used to interact with the code document, thus ensuring the document is always syntactically correct.

The approach that is envisioned here loosens this tight MVC concept a bit in order to give the user more freedom in typing, though ensuring the synchronisation to the model. A multi-step, incremental update process is proposed instead. This process relies on the representation of the textual document as a model of syntactic constructs. A metamodel for this representation is presented in the next section.

4.2 Textblocks Meta-model

The basic idea of this text-blocks metamodel is based on the decorator pattern. Prominent examples where this pattern is also successfully used are graphical syntaxes for metamodels. Most graphical approaches handle their diagrams as models, too. This means that there is a generic meta-model for diagrams which could be called a “diagram meta-model”. Instances of this metamodel are then graphical elements such as rectangles or arrows. The decorator pattern comes into play as these instances decorate (non-intrusively reference) elements from the main model (also called “domain-model”, as it represents the actual data). Hence, what a diagram model actually does is to describe how a specific domain-model is represented as a diagram.

To be able to decorate an arbitrary model as a textual view, a meta-model was developed that can represent generic text structures. As shown in figure 1 the textual view is then in fact a view on the decorated domain model. The user still types text as in any other development environment but what happens is that this text is actually editing the text-blocks and the domain model.

Figure 2 depicts this metamodel. As the structure of text is rather simple the meta-model only contains a few different constructs. The main class of this metamodel is, after its name, the `TextBlock` class. A `TextBlock` represents something like a meaningful unit of work within the text. For example, this could for a Java like language be a method body. These `TextBlocks` can then have nested `TextBlocks` as their children. In the Java example these would represent correspondingly nested units within the method body, as for example a while loop or an if-block.

The correspondance between these `TextBlocks` and the actual constructs from the metamodel as well as the syntactic constructs is defined using the `TextBlockDefinition` class. For our approach to work we assume that the actual concrete to abstract syntax mapping is also available as a mapping model. Such models are used in many approaches [12][13][17] that are used for the definition of a textual syntax for

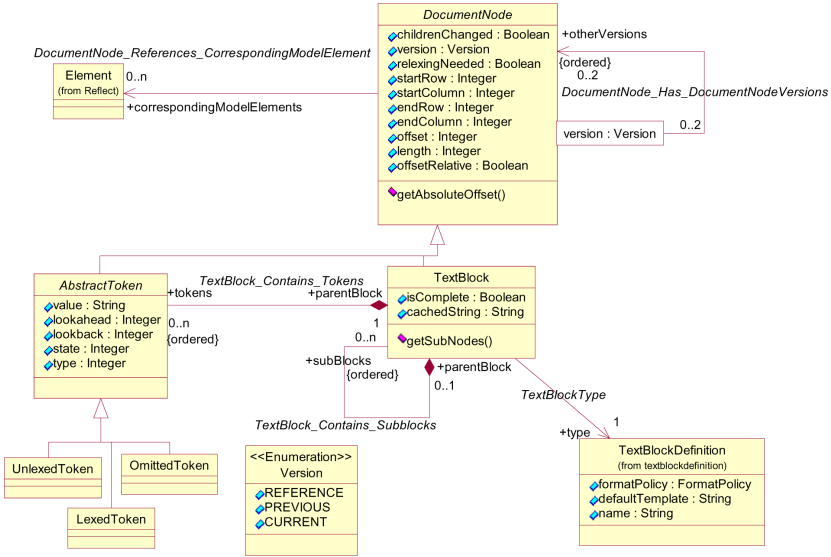


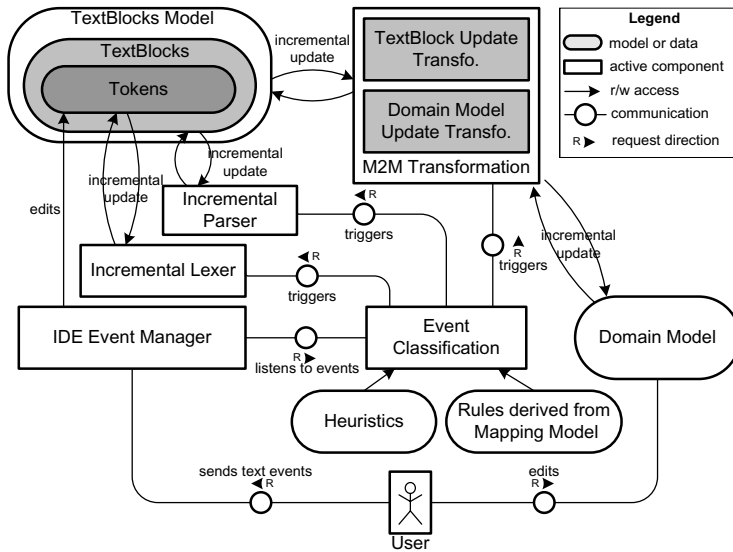
Fig. 2. TextBlocks metamodel

metamodels. The `TextBlockDefinition` then references the corresponding construct from this mapping model. This correspondence defines the *type* of a `TextBlock`. Thus, it is possible to see which rule from the mapping was used to create a particular `TextBlock`. This information can then later be used to identify elements that can be re-used if the internal structure of a `TextBlock` changed during editing by the user (See next Section 4.3 for more details on this).

Summarising, concerning an update process from text to the domain-model a text-blocks model makes the textual information accessible for model transformation that turn text parts into valid model elements of the domain-model. As it can be seen in the meta-model each `TextBlock` element may reference corresponding model elements for which it is used as decorator for the textual view.

4.3 Incremental Update Approach (Text → Model)

The incremental update process including its driving components is depicted in Figure 3. All textual editing that is done by the user is first captured as event by the event manager of the IDE. A central component within the presented approach is the *Event Classification* component. This component decides what to do with a sequence of user events (in this case these events are textual editing events, meaning inserting or deleting characters). The classification component then triggers the incremental update components accordingly. For instance, upon a completion of an edit of a terminal token (e.g., because the user changed the cursor position to somewhere else in the document) the classification component triggers the incremental lexer to check whether tokens were changed, added or deleted. Depending on the outcome of the incremental lexing process the incremental parser may be triggered to check if the structure of



the text-blocks document has to be changed due to the changed tokens. As a next step a model-to-model (M2M) transformation may be triggered that updates the domain model correspondingly.

As it is also possible to directly edit the domain model (e.g., through a refactoring command) the synchronisation transformation can also be called to update the text-block model correspondingly.

As the user actually directly works on the text-blocks model any text event is directly passed to this model. In fact, edits that are only local changes e.g., within a terminal token the value token gets directly changed. Therefore also all corresponding model elements which depend on this token will get changed accordingly.

Wagner [4] presented an efficient incremental update approach that includes self-versioning documents. In Wagner’s approach these documents support the incremental lexing and parsing process. This means, whenever the user edits the document, the corresponding node within the document becomes a new version of that node. The information of the new version compared to the already lexed and parsed reference version that is still kept is then used to decide how the incremental lexing/parsing needs to be done. Our meta-model also incorporates the capabilities of these self-versioning documents. We also implemented the incremental lexing approach that is presented there. Further steps such as incremental parsing and transformation to the domain-model are still subject to research within our prototype.

4.4 Event Classification

It is essential for this approach to detect the actual intention of the user when he types something in order to change the textual representation of a model. Consider, again

a Java like language. For example, if a user starts to delete the terminal tokens that belonged to a method definition, it has to be identified at what point in time the actual method model element will be also deleted. There are multiple possibilities for that, starting from the deletion of only the actual method identifier to the deletion of the whole section including the body of the method. As the possible intentions heavily depend on the characteristics of each language the event classification mechanism that we want to employ is multi-fold. We categorize the actions across several categories or dimensions.

1. **ID-preserving actions:** Actions within this category are changes that preserve the id of their corresponding model element. For example, the addition of an inner element will mostly fall into this category. This category can be split into several more sub-categories:
 - (a) **Refactoring actions:** Actions that are explicitly called to make a change that is meant to be ID-preserving. For example, in a Java like language there might exist a refactoring operation that explicitly alters a method signature including its name and parameters. Because this changed was called as a refactoring operation the intention of the user is made clear that he wants to keep the identity of the operation.
 - (b) **Local change actions:** Actions that are defined as a local change are not required to undergo an ID-preserving update mechanism. No additional tries are made to preserve any of the elements upon a severe change. Elements that are modified within this kind of actions can then simply be deleted and re-created.
 - (c) **Structure-altering actions:** These actions only change the structure of the elements and do not invalidate their ID. For example, a whole block is moved within the text resulting on the model side in a re-ordering of elements.
2. **ID-destroying actions:** These actions will change an element in a way that its ID cannot be preserved. For example, the deletion of a part of the text that represents an element where the same text is typed in again later on. This should, though ultimately depending on the language, in most cases be meant as reviving the old element which is already lost.

In some cases it is possible that an action cannot be uniquely assigned to one of the categories. Therefore, it is necessary to configure the editor's behaviour to use one possibility as default and make the other alternative available as explicitly triggerable action (as for example the refactoring actions in Eclipse JDT [18]). Alternatively it might be possible to consult the user, each time such a change is done, asking what he actually intended.

There are also different ways how to classify different possible editing actions into these categories:

1. A default set of event classifications can be pre-defined based on the general structure of the lexing and parsing mechanism. For instance, a change within the lexeme of a terminal token that does not destroy its type is, by default, an ID-preserving action. Also handling of additional lexical information such as format or documentation artefacts can be handled like this by default.

2. As already introduced in related work ([6]), one part of the available change actions can directly be derived from the grammar. These so called *syntax-directed* editing actions check what are the possible next productions at a certain point within the text and provide them as auto-completion like actions.
3. Default semantical heuristics are provided based on the mapping definition that specify how certain elements should be treated. For example a standard behaviour may define what should happen to elements that were recognized by a certain grammar production when a different production is applied to those existing elements after a change. One possibility would be to keep all elements that can heuristically be identified as elements of the same type of a previous previous production rule identification.
4. User defined semantical actions can additionally be defined to the actual mapping definition to classify a certain change set having a certain intention.

5 Conclusion and Future Work

In this short paper we envisioned an approach to handle concrete textual syntaxes for models on top of a UUID-based repository. We intend to employ existing algorithms for incremental lexing and parsing in combination with additional rules and heuristics that help to identify the actual intention behind a user's editing operation.

We are currently implementing a prototype that is based on the Eclipse IDE. For the current prototype implementation we chose the TCS approach [12] as basis. The parser used in TCS is the ANTLR parser generator which we additionally enhanced with incremental lexing capabilities. Furthermore, we implemented a custom subclass of the `ITextStore` interface, which is used by eclipse text editors to interact with a document on which they work. This custom implementation provides a view on a textblock model instead of a simple text file. Our implementation maps all operations that are supported on the text store directly to a textblocks model. However, it is still possible to store the document as text through an export functionality of this textblocks document.

As a next step we will implement incremental parsing techniques. Furthermore, we will employ the presented event/actions classification to a case study project to see what actions can be identified and which incremental updates need to be triggered when one of these actions is identified.

References

1. Object Management Group: Object Constraint Language (OCL) Specification Version 2.0. OMG Document No 05-06-06
2. Uhl, A.: Model-driven development in the enterprise (2007) (Last retrieved 2008-07-06), <http://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/7237>
3. Uhl, A.: Model-driven development in the enterprise. *IEEE Software* 25(1), 46–49 (2008)
4. Wagner, T.A.: Practical Algorithms for Incremental Software Development Environments. PhD thesis, University of California, Berkeley (1998)
5. Nagl, M. (ed.): Building tightly integrated software development environments: the IPSEN approach. Springer, New York (1996)

6. Teitelbaum, T., Reps, T.: The cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM* 24(9), 563–573 (1981)
7. Bahlke, R., Snelting, G.: The psg - programming system generator. *SIGPLAN Not.* 20(7), 28–33 (1985)
8. Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: Bruel, J.-M. (ed.) *MoDELS 2005*. LNCS, vol. 3844, pp. 159–168. Springer, Heidelberg (2006)
9. Karlsch, M.: A model-driven framework for domain specific languages. Master's thesis, University of Potsdam, Hasso Plattner Institute (2007)
10. Garcia, M., Sentosa, P.: Generation of Eclipse-based IDEs for Custom DSLs. Technical report, Software Systems Institute (STS), Technische Universität Hamburg-Harburg, Germany (2007)
11. Krahn, H., Rumpe, B., Völkel, S.: Efficient editor generation for compositional dsls in eclipse. In: *Proc. of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM 2007)*, Montreal, Quebec, Canada (2007)
12. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: *GPCE 2006*, pp. 249–254. ACM Press, New York (2006)
13. Scheidgen, M.: Textual editing framework (2007) (Last retrieved 2008-07-06), <http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/tool.html>
14. Efftinge, S.: Xtext reference documentation (2006) (Last retrieved 2008-07-06), http://www.eclipse.org/gmt/oaw/doc/4.1/r80_xtexReference.pdf
15. Eclipse Foundation: The Eclipse Modelling Project (2006) (Last retrieved 2008-07-06), <http://www.eclipse.org/modeling/>
16. Goldschmidt, T., Becker, S., Uhl, A.: Classification of Concrete Textual Syntax Mapping Approaches. In: *Proc. of the 4th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA 2008)*, pp. 169–184 (2008)
17. Fondement, F.: Concrete syntax definition for modeling languages. PhD thesis, Ecole Polytechnique Fédérale de Lausanne (2007)
18. Eclipse Foundation: Eclipse Java Development Tools (JDT) Subproject (2008) (last visited 24.01.2008), <http://www.eclipse.org/jdt/>

A Model Engineering Approach to Tool Interoperability

Yu Sun¹, Zekai Demirezen¹, Frédéric Jouault², Robert Tairas¹, and Jeff Gray¹

¹ Department of Computer and Information Sciences, University of Alabama at Birmingham
{yusun, zekzek, tairasr, gray}@cis.uab.edu

² AtlanMod (INRIA & EMN), Nantes, France
frederic.jouault@inria.fr

Abstract. The integration of various tools is a common requirement throughout the software development process. It is often desirable to consult several tools that perform similar functionalities in the same domain to obtain different perspectives and results to assist design and maintenance decisions. In many cases, tool interoperability requires the generalization of tool-specific data, which necessitates homogenizing the data such that intellectual assets can be shared through a common framework (e.g., the integration of results from various clone detection tools). This tool demonstration summary presents a software language engineering solution technique that uses Model-Driven Engineering to address tool interoperability. A specific focus of the paper is a demonstration of model transformation applied to the task of homogenizing different data formats among similar tools. The challenges of tool integration are discussed in the paper, along with a detailed case study that highlights the benefits of applying a model transformation solution to tool interoperability.

Keywords: Model Engineering, Tool Interoperability, Model Transformation, Domain-Specific Languages, AMMA.

1 Introduction

The ability to model the key characteristics of new tools, and to integrate them with a set of previously defined tools, can be very useful. Often, however, researchers independently develop similar tools to perform the same functionality within a particular domain. Each isolated effort defines a different semantic model and uses uncommon storage representations. Unfortunately, this poses a problem when it comes to the important issue of integration – the result is an inability to provide a seamless exchange between tools. For example, our own past work observed numerous tools within the domain of avionics fault analysis that had very much in common, but those tools could not share models. A solution was an integrated model and associated tool adapters that allowed data exchange among the tools [7]. The area of Enterprise Application Integration (EAI) has provided several additional technologies to assist in the tool integration problem across many domains, such as healthcare [19]. This tool demonstration paper summarizes our investigation into using concepts from software language engineering, in particular model transformation, to address the tool integration problem.

In tool interoperability, different standards and formats make software interoperability a challenge [2]. There are several existing approaches that can be adopted to overcome the limitations of tool interoperability. For example, a general approach that is based on traditional parsing and interpreting activities can be implemented with general-purpose programming languages. As a second example, XML-based interoperability has emerged as a popular choice for a generic exchange format between software tools. However, although this works relatively well when all considered tools use some form of XML, this kind of solution is not as convenient in other contexts (e.g., when context-free parsing of the storage format is necessary).

A third approach to interoperability is based on model transformation and is the focus of this paper: the different formats are captured as abstract definitions of data structures (i.e., metamodels), and transformation rules map from one representation to another. The AtlanMod Model Management Architecture (AMMA) [8] is a model engineering framework that may be used to build bridges between tools or Domain-Specific Languages (DSLs). Each tool or DSL is captured and represented as a coordinated set of models. The work described in this tool demonstration summary uses three of the main capabilities of AMMA: metamodeling with the Kernel MetaMeta-Model (KM3) [4], model transformation with the AtlanMod Transformation Language (ATL) [6], and projections to (i.e., extraction) and from (i.e., injection) other technologies (e.g., grammars, XML). Projections are especially useful in the context of model-driven tool interoperability because each tool typically uses a specific file format. Notably, AMMA provides the Textual Concrete Syntax (TCS) [5] tool to deal with context-free syntax.

Although our specific tool demonstration is focused on using AMMA as a solution strategy, we believe that the general concept can be used with most modeling and language engineering tools. The next section presents a case study that demonstrates how software language engineering (specifically model engineering and model transformation) can be used to assist in the sharing of results and data across tools from the same domain. A concluding section summarizes the paper by presenting lessons learned and pointing toward future work.

2 Visual Representation for Clone Detection

Code clones are blocks of statements that are duplicated in multiple locations of one or more programs. Programs containing code clones are prone to manifest problems in the maintenance phase of the software development process [13]. A number of tools have been developed to automatically detect code clones in programs, such as CloneDR [1], CCFinder [12], Simian [10], and SimScan [11]. These tools take the source program as input and generate mostly textual reports about the code clones that were detected. Figure 1 shows excerpts of the reports generated by three popular code clone detection tools. The reports record the location and length of each code clone. Depending on the size of the application that is analyzed, the corresponding clone report files could contain tens of thousands of lines of text.

Simian	Found 6 duplicate lines in the following files: Between lines 117 and 122 in E:\source\FieldRecordTextTest.java Found 6 duplicate lines in the following files: Between lines 611 and 617 in E:\source\Utility.java Between lines 602 and 608 in E:\source\Utility.java
SimScan	34751-E:\source\Utility.java:107-110, 34912-E:\source\Utility.java:119-122, 11421-E:\source\ExtendedVector.java:35-38, 34806-E:\source\Utility.java:111-114, 34967-E:\source\Utility.java:123-126, 48713-E:\source\WeightedStatistics.java:60-63
CloneDR	#1 5a7e550 +1234 rule_name 0.970 2 2 11 #2 5a7e550 3bcb780 19 0 29 0 E:/source/Utility.java #2 5a7e550 3bd6560 45 0 55 0 E:/source/KeyCounterTest.java

Fig. 1. Excerpts of three code clone detection reports

2.1 Challenges of Comprehending Multiple Tool Results

Because many tools use different criteria and methods to detect clones, it is desirable for users to observe the reports of several tools and compare them so that a more comprehensive understanding of the code clones can be obtained. However, these tools apply varying formats and representations in their reports, making it very challenging to understand, compare, and integrate the results from these diverse sources. Therefore, a uniform format for code clone detection reports to which different outputs could be automatically transformed would greatly simplify the integration process of various detection results. Moreover, a graphical representation of clones could further aid clone comprehension.

2.2 Overview of Solution Approach Using Model Engineering

This section introduces our work that transforms text-based code clone detection reports to SVG (Scalable Vector Graphics) [9], a language that describes two-dimensional graphics in XML. More specifically, certain shapes and graphical components will be drawn to represent the code clone information according to the textual results. The reason for choosing SVG as the final visual representation is that the language is declarative, which provides for a simpler metamodel. In addition, because SVG is based on XML, the built-in XML facilities of AMMA can be used.

Using our model transformation approach to tool interoperability, the reports in Figure 1 will be transformed to SVG code and displayed in a web browser. Performing source-to-source transformations can realize this, but most of the process would need to be hard-coded and therefore too complex and error-prone to adapt and extend when new clone detection tools emerge. As an alternative to source transformation, we use model engineering with a source-to-model-to-source approach, where the first source refers to context-free text, and the second corresponds to XML.

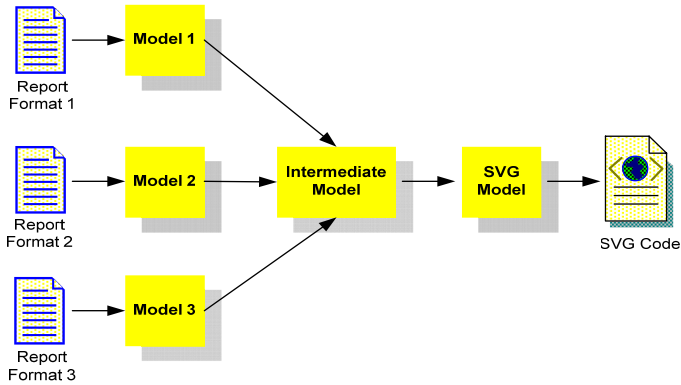


Fig. 2. Overview of code clone detection tools interoperability

Because models have a more structural syntax to capture the main concepts and relationships (i.e., models are primarily focused on abstract syntax), implementation of tool data mappings can be simplified through model transformations. In our approach, we first project the textual report into its corresponding model through injection with TCS. This source model is then transformed into the target SVG model. Finally, we project SVG code from the SVG model through extraction. Figure 2 provides an overview of our approach. Our goal is to generalize the experience gained in this case study of one domain to address tool interoperability problems in other domains. Usage of an intermediate model makes it possible to decouple the transformations dealing with the various source formats from the transformation that produces the SVG model.

2.3 Implementation of Clone Detection Tool Interoperability

The basic idea of the model-driven solution is to define the abstract syntax (meta-model) and concrete syntax (grammar) for each of the different code clone detection reports and SVG, then execute associated transformations between the code clone detection reports domain and the SVG domain. These two steps can be performed with the facilities provided by AMMA. Figure 3 shows a lower level view of Figure 2 and offers the detailed structure of the implementation. Each step will be explained in the following sub-sections.

Definition of Sub-domains and Injection of Source Reports. Because the transformation process operates on models, the very first step involves injecting the textual code clone detection report generated by a certain tool into its corresponding model. To realize this, the metamodel must be defined for the tool's report using KM3 (Step 1 in Figure 3). Figure 4 shows the metamodel designed for the detection report

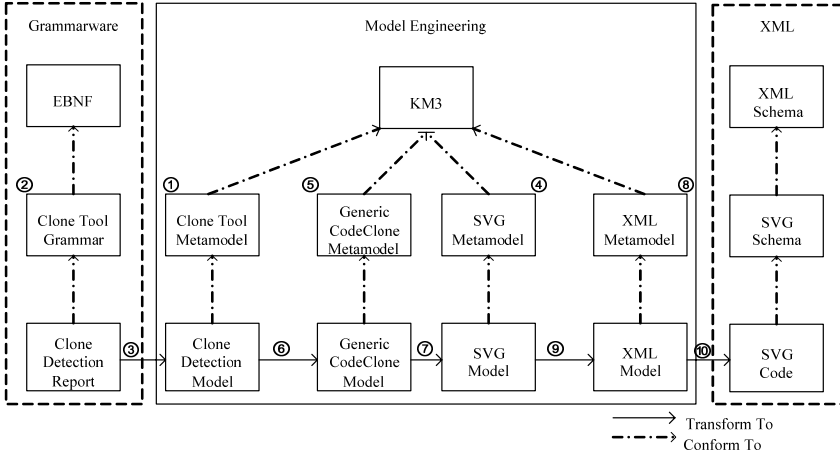


Fig. 3. Overview of the code clone tool interoperability implementation

generated by Simian¹. The report includes summary information about the analyzed files (e.g., the total number of lines checked, total number of blocks, and clone pairs). This metamodel represents the abstract syntax, because it defines the components and their relationships without the information about the concrete grammar.

Parsing is needed to translate the textual report and capture the necessary information. By defining the context-free grammar for a tool report using TCS (Step 2 in Figure 3), the textual syntax is mapped to its metamodel, so that it can parse the input file and inject it into the model, which conforms to the metamodel. However, not all reports are based on a context-free grammar. This necessitates some parts of the reports to be preprocessed for correct parsing.

For each kind of code clone detection tool or sub-domain, both a metamodel specification in KM3 and a concrete syntax definition in TCS are needed. These specifications work together to enable injection of the initial detection reports into models for the later transformation process (Step 3 in Figure 3).

Definition of Target Domain and Transformations. When the input model is ready, it can be transformed to the target model. In this experiment, SVG is the target domain, so the SVG metamodel must be defined first in KM3 (Step 4 in Figure 3). The TCS specification for the SVG grammar is not needed, because SVG is based on XML and AMMA provides an embedded extraction engine to generate XML. More details about this will be given in the next step. SVG is an XML-based language that enables the specification of a graph containing rich elements, so a complete metamodel for SVG can be very large. Because only a small number of elements (e.g., group, rectangle, and text) will be used in this interoperability phase, a subset of the SVG metamodel is used to satisfy these requirements, as shown in Figure 5.

¹ Please note that due to space considerations, the metamodels presented in this paper are UML class diagrams. The actual KM3 metamodels and all other artifacts of the project are available at [15].

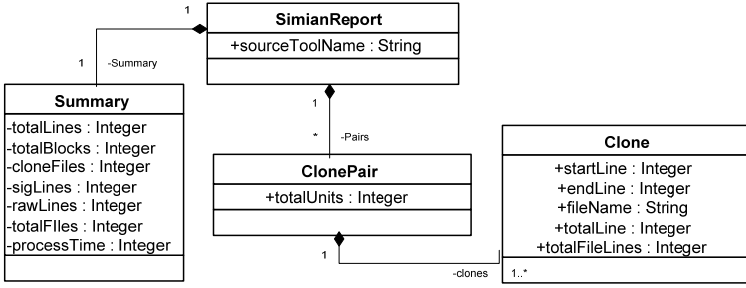


Fig. 4. Metamodel for Simian's code clone detection report

Our goal is to transform multiple tool results from the code clone domain to the SVG domain. However, defining the transformation process for every tool to SVG is not efficient and extensible, because these are two totally different domains with little direct connections and mappings; hence, the transformation rules would be comparatively complicated. If it is required to transform each code clone detection report to SVG directly, multiple complex transformations between these two domains would be required. In order to optimize the whole process, we defined an intermediate model called a *Generic Code Clone* (GCC) model (conforming metamodel in Figure 6), which is closely related to the code clone domain that contains the common concepts and features of code clone detection tool reports (Step 5 in Figure 3). This is actually the minimum set of the features shared by all the tools. Instead of transforming each Clone Detection model to the final SVG model, the Clone Detection model is transformed to the GCC model (Step 6 in Figure 3), which is then transformed to the SVG model (Step 7 in Figure 3). In this way, only one transformation from the code clone domain to the SVG graphical domain is needed. A similar idea also appears in compiler design, where an Intermediate Representation (IR) is used to optimize code generation [16].

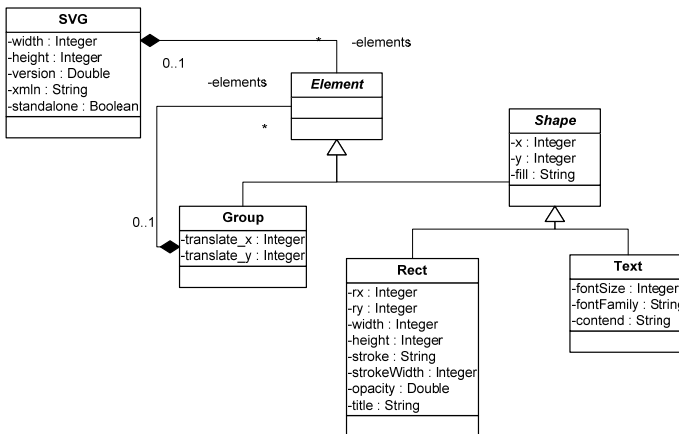


Fig. 5. Metamodel for a subset of SVG

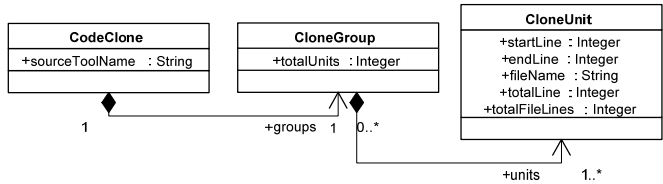


Fig. 6. Generic Code Clone (GCC) metamodel

The principles under the transformation from the GCC model to the SVG model are the following:

- A rectangular box is drawn for each clone group.
- Smaller rectangles are drawn inside this box to represent every file that contains one of the clone units in the clone group. The length of these rectangles is based on the length of the file they each represent.
- Colored lines with different width are used to represent the location and length of the code clones in the files.

We use ATL to write the transformation rules, specifying the mappings of the elements between the two domains.

Generation of Target Code. An extraction process to generate the SVG code from an SVG model is needed. The AMMA platform supports automatic generation of XML from an XML model. Because SVG is based on XML, in the final step, the transformation is defined in ATL from the SVG model to the XML model (Step 9). However, the XML metamodel must be defined to enable the transformation (Step 8). When the final XML model is produced, the SVG code can be generated by a single extraction (Step 10).

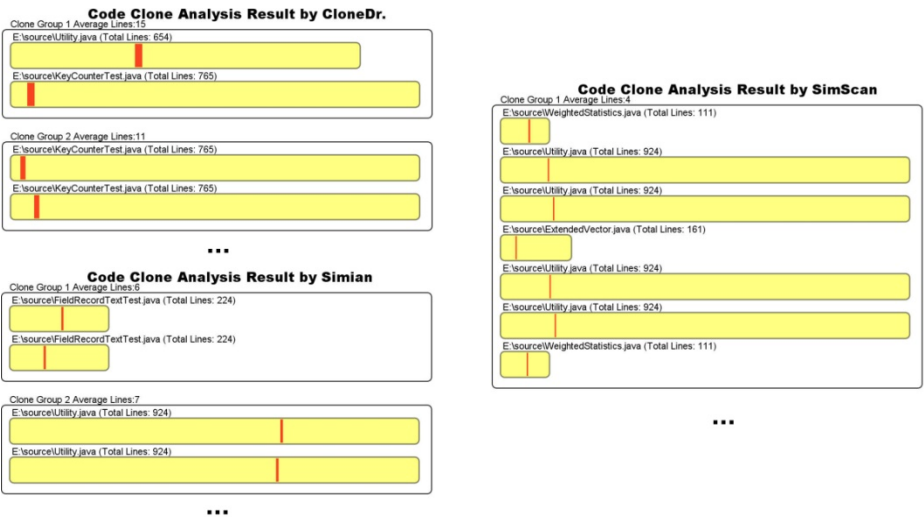


Fig. 7. Uniform visual representation for reports of the three tools (subset of reports)

Interoperability Results Reported by SVG. Figure 7 shows the result of one conversion experiment to SVG. The three small graphs correspond to the three textual reports in Figure 1. The files are represented by rectangles with different lengths. The location and length of each code clone block is highlighted in the rectangle. Also, other information such as the file name and average number of clone lines is shown. Support for other tools (e.g., CCFinder [12]) can be extended by: 1) defining the new tool's metamodel and textual grammar, and 2) writing the transformation rules from the new tool model to the GCC model. The complete experiment can be found in [15].

3 Conclusion and Future Work

The challenge of tool interoperability often occurs in the development of applications where a cadre of analysis and design tools is available. Each specialized tool contributes to a crucial step in the development process. It would be beneficial to capture the information in the context of one tool and use it in a different tool. We have used software language engineering to implement one form of interoperability which concerns the data exchange and sharing of information between tools. There are several lessons that were learned from applying model transformation to the tool interoperability problem:

- **Model transformation provides separation of concerns across the integration process:** Separation of concerns in our experiments is shown in four aspects: 1) The whole process is composed of three parts – defining source domains, defining target domain, and writing transformation rules; 2) When defining a domain, the metamodel (abstract syntax) and parser (concrete syntax) are specified separately. The AMMA platform connects the abstract syntax and concrete syntax when doing injection; 3) Metamodels have a clear and organized structure about the components and relationships they contain, which enables users to focus more on the concept and semantic mappings when writing transformation rules.
- **Adaptability and extensibility in defining new tools:** It is often necessary to define a new tool or DSL when implementing tool interoperability. We observe that a certain level of modularity may be achieved by introducing intermediate (or *pivot*) metamodels in the transformation chain between the source and target domains. For instance, a generic or pivot tool is often needed to optimize the exchange among different tools. In the context of model transformation, only a metamodel is needed for a new tool. Although some tools with complex functions may make the metamodel difficult to specify, we can define a subset of the tool to satisfy the specific need of a tool interoperability case. In some other cases, a parser (concrete syntax) is also needed to support a new tool. In this situation, the extensibility depends on the complexity of the grammar, because building a parser is not an easy task [17]. However, because most of the tools are domain-specific, their grammar and parser are generally much simpler than general-purpose programming languages. In addition, some preprocessing can simplify the complex detection report text in order to make the parser easier to build. Admittedly, if the tool or data has a very complex syntax that makes building the parser extremely difficult, additional effort is needed.

- **Models should be the primary representation for tools:** Models are not the initial and final representations for most tools, so an extra initial effort is needed to inject and extract the tool data into a model. This involves multiple steps, and might make the whole approach less direct and efficient than source-to-source transformation. Fortunately, some powerful facilities are provided to simplify the exchange between sources and models. For example, in AMMA, XML source code can be automatically generated from the XML model by an embedded XML engine. In addition, the Ant build tool [18] is also supported in AMMA, which enables the generation of build files to execute all the steps automatically.
- **Maturity of modeling tools:** Although tools like AMMA provide many benefits of software language engineering, there are still limitations in the support and usage of such tools. As an example, the level of debugging support in AMMA is not as mature as in traditional programming languages. Furthermore, we believe that the general principles of using software language engineering to solve the tool interoperability problem is not unique to AMMA; other language environments should also be able to provide similar advantages.

As implied by the last lesson learned, there are several topics that represent future work in this area. One limitation of this approach is that the number of tools that can be integrated depends on the ability to parse the tool representation. The textual format of some tools may not be based on a context-free grammar, which complicates the mapping process. The only option is to eliminate the parts that do not conform to the context-free grammar. However, this will make the whole process less automated, and will also make it possible that some important information is lost.

The complete details about the project (e.g., full source for the various KM3, TCS, and ATL specifications) are available at the project website [15].

Acknowledgements. This work was supported in part by an NSF CAREER award (CCF-0643725), an NSF CPA award (0702764), the OpenEmbeDD project, and the EDONA project.

References

- [1] Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone Detection using Abstract Syntax Trees. In: International Conference on Software Maintenance, Bethesda, Maryland, pp. 368–377 (1998)
- [2] Benguria, G., Larrucea, X.: Data Model Transformation for Supporting Interoperability. In: International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems, Alberta, Canada, pp. 172–181 (2007)
- [3] Budinsky, F., Steinberg, D., Ellersick, R., Merks, E., Steinberg, D., Grose, T.: Eclipse Modeling Framework. Addison-Wesley, Reading (2003)
- [4] Jouault, F., Bézivin, J.: KM3: A DSL for metamodel specification. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 171–185. Springer, Heidelberg (2006)
- [5] Jouault, F., Bézivin, J., Kurtev, I.: TCS: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: International Conference on Generative Programming and Component Engineering, Portland, Oregon, pp. 249–254 (2006)

- [6] Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Model Transformations in Practice Workshop, International Conference on Model-Driven Engineering, Languages, and Systems, Montego Bay, Jamaica (2005)
- [7] Karsai, G., Gray, J.: Component Generation Technology for Semantic Tool Integration. In: IEEE Aerospace Conference, Big Sky, Montana, pp. 491–499 (2000)
- [8] Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL Frameworks. In: International Conference on Object-Oriented Programming, Systems, Languages, and Applications, Portland, Oregon, pp. 602–616 (2006)
- [9] Scalable Vector Graphics (SVG), <http://www.w3.org/Graphics/SVG/>
- [10] Simian, <http://www.redhillconsulting.com.au/products/simian/>
- [11] SimScan, http://www.blue-edge.bg/simscan/simscan_help_r1.htm
- [12] Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A Multi-Linguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering* 28, 654–670 (2002)
- [13] Tairas, R., Gray, J.: Phoenix-Based Clone Detection Using Suffix Trees. In: ACM South-east Conference, Melbourne, Florida, pp. 679–684 (2006)
- [14] Tratt, L.: Model Transformations and Tool Integration. *Software and Systems Modeling* 4, 112–122 (2005)
- [15] Representation for Clone Tools (Eclipse ATL Use Case), <http://www.eclipse.org/m2m/at1/usecases/VisualRepCodeClone>
- [16] Aho, A., Lam, M., Sethi, R., Ullman, J.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston (2007)
- [17] Lämmel, R., Verhoef, C.: Cracking the 500-Language Problem. *IEEE Software* 18, 78–88 (2001)
- [18] Apache Ant, <http://ant.apache.org>
- [19] Khoubati, K., Themistocleous, M., Irani, Z.: Investigating Enterprise Application Integration Benefits and Barriers in Healthcare Organisations: An Exploratory Case Study. *International Journal of Electronic Healthcare* 2, 66–78 (2006)

Engineering Languages for Specifying Product-Derivation Processes in Software Product Lines^{*}

Pablo Sánchez¹, Neil Loughran², Lidia Fuentes¹, and Alessandro Garcia²

¹ Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga, Málaga, Spain
{pablo,lff}@lcc.uma.es

² Computing Department, InfoLab 21
South Drive, Lancaster University, LA1 4WA, UK.
{loughran,garciaa}@comp.lancs.ac.uk

Abstract. The goal of a Software Product Line (SPL) is to provide a set of reusable software assets for the rapid production of a software systems family aimed at a specific market segment. The main objective of SPL engineering is to construct, as automatically as possible, specific products after selecting the particular set of features that must be included in them. Unlike traditional engineering of single systems, SPL engineering often requires dealing with three different languages at each stage of the software lifecycle: (1) a language for specifying the variability of the SPL (e.g. a feature model); (2) a language for designing the reusable software assets (e.g. UML 2.0); and (3) a language that specifies how these reusable assets must be composed for constructing specific products. There are currently available enough languages for variability specification and software assets design, but there is a general lack of languages for specifying and automating the composition of these assets. This paper presents as a novel contribution a process to engineer this kind of language. The process produces an editor and a “compiler”, which automates the composition of reusable assets, for a particular language. To explain this process a language has been developed to compose reusable assets of architectural models.

1 Introduction

A Software Product Line (SPL) [1,2] aims to provide the infrastructure for the rapid production of similar software systems for a specific market segment. These systems share a subset of common features, but there are also some variations between them. One of the main goals of a software product line is to, as automatically as possible and in a prescribed way, construct specific products after a set of choices and decisions has been adopted, specifying which variants must be

^{*} This work has been supported by EC Grants IST-2-004349-NOE AOSD-Europe and AMPLÉ IST-033710.

included. Software product line engineering, as compared to traditional engineering of a single software system, introduces new challenges such as: (1) *variability specification*; (2) *variability design* and (3) *variability composition specification*, also called *product derivation*.

Variability specification deals with the identification of which are the common and variable features of a family of products, the conditions that make them variable (e.g. they are optional features), as well as the constraints and dependencies between them. *Variability design* is concerned with the creation of software specifications or designs that incorporate mechanisms, such as conditional compilation, parameterization or plug-in components, for supporting the variations inherent in a family of products. *Variability composition specification* or *product derivation* focuses on the definition of rules that specify how reusable software assets of a family of products must be composed in order to obtain specific products of that family.

Whereas there is wide range of languages for variability specification and variability design, there is a general lack of languages for specifying product derivation processes. Some tools, such as pure::variants [3], Gears [4], exist at the implementation level for specifying composition of implementation level artefacts, but they may be difficult to adapt to deal with languages used at other development stages, such as UML.

With the emergence of model-driven development techniques [5], it is commonly accepted that model transformations are an effective mechanism for the automation of product derivation processes at different modelling levels. Nevertheless, the specification of product derivations with a general-purpose model transformation language, such as ATL [6] or xTend [7], forces software product line engineers to learn intricate details of model-driven techniques. For instance, as these languages are often based on abstract syntax manipulations, they require detailed knowledge of the metamodel where the software product line model is expressed (i.e., the metamodel of the variability design).

In order to overcome these problems, we propose to create specific languages for specifying product derivation processes for each software development stage. These languages use abstractions and mechanisms relevant to that development stage. Thus, these dedicated languages use terminology familiar to engineers working at that abstraction level. These high-level specifications of product-derivation processes are then automatically “compiled” to a set of model transformations, expressed in a general-purpose model transformation language. This low-level model transformation automates the product derivation process. The compilation process hides low-level details and intricacies of general-purpose model transformation languages from software product line engineers, allowing them to work at a higher abstraction level. We present in this paper, as a novel contribution, a process for engineering this kind of language and constructing “compilers” for it.

This process is applied to the engineering of a language for specifying product derivation processes at the architectural level. The created language is applied to

an architectural model of a Smart Home system, which is a typical benchmark for SPL architecture modelling [8,1].

This paper is structured as follows: Section 2 describes the motivation for engineering dedicated languages for specifying product derivation processes in software product lines. Section 3 explains the process for engineering these dedicated languages. Section 4 provides the discussion and a comparison with related work. Section 5 outlines some conclusions and future work.

2 Our Approach

This section explains the motivation behind this work using as an example the specification of a product derivation process for the architectural model of a lock control framework for a Smart Home software product line [1,8] (Section 2.1). This case study is used to discuss the shortcomings of using general-purpose model transformation languages for specifying product derivations (Section 2.2) and the benefits of creating dedicated product derivation languages (Section 2.3).

2.1 Lock Control System in a Smart Home Software Product Line

Smart Home applications aim at automating and controlling houses and buildings in order to improve the comfort and security of their inhabitants. As part of a Smart Home SPL, a door lock control framework must be designed [1]. This lock control is placed on doors of rooms whose access must be controlled. Several options are available to end users acquiring a specific Smart Home software installation:

- Different authentication mechanisms can be used: identification cards, fingerprint scanners or a simple numeric keypad.
- Doors are opened manually and users have a time period to authenticate before triggering the alarms. Optionally, it is possible to select a computer-controlled door lock control (Automatic Lock), which will be released upon successful authentication.
- Optionally, sliding doors that open automatically can also be used (Door Opener). This option requires that the Automatic Lock control of the door lock be selected.

We explain the necessity of engineering languages for product derivation using an architectural design for this lock control framework, which is illustrated in Fig. 1. As already commented in the introduction, this architectural design is comprised of three models.

Firstly, variability inherent to the domain is expressed using a cardinality-based feature model [9] (Fig. 1 (top)). This feature model represents *variability specification* or *solution space*. It specifies which features of the system are variable and the reasons why. For instance, the `AuthenticationDevice` to be used is a variable feature because there are several devices available but only one must

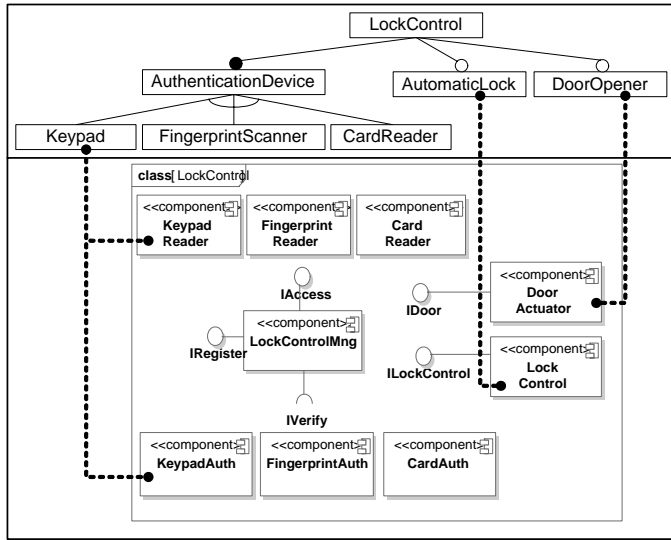


Fig. 1. Reference architecture for the lock control framework: variability (top) and component (bottom) views

be selected. **AutomaticLock** and **DoorOpener** are variable features because they are options that may be included in a specific lock control application or not.

Secondly, once variability has been identified, the architecture is designed using the component model of UML 2.0 (Fig. 1 (bottom)). This represents *variability realisation* or *problem space*. The mechanism selected for supporting variability in the architectural design is *plug-in* components. The **LockControlMng** component is the central component of this architecture. Each alternative for authentication is designed as a pair of plug-in components: one for controlling the physical device that serves to authenticate users (e.g. **KeypadReader**); and the other one encapsulating the logic of the authentication algorithm (e.g. **KeypadAuth**). These plug-in components communicate with the **LockControlMng** through the **IAccess** interface, in the case of reader components, and the **IVerify** interface, in the case of authenticator ones. All plug-in components must register in the **LockControlMng** component using the interface **IRegister**.

The **LockControlMng** receives data from the reader components and, with the data received, it calls the authenticator component. The latter is in charge of checking if the user has access to the room or not. If the user is authentic, the **LockControlMng** component invokes the **LockControl** component, which releases the lock. This invocation is placed only if the automatic lock control option has been selected. If the door is a sliding one, the **LockControlMng** should also invoke the **DoorActuator** component for automatic opening of the door.

Thirdly, we must specify the links between *variability specification* and *variability design*, or *problem space* and *solution space*, indicating how the components of the architectural model must be composed according to the selected features.

```

00 deriveProduct(uml::Model inputModel, LockFmkCfg::LockFramework cfg):
01   inputModel.setName(cfg.name) ->
02   inputModel.configureAuthentication(cfg.authenticationDevice) ->
03   cfg.automaticLock?inputModel.addAutomaticLock():
04   inputModel.removeAutomaticLock() ->
05   ....
06 addAutomaticLock(uml::Model model) :
07   model.selectElement('LockControlMng').createRequires(
08     model.selectElement('ILockControl'))->
09   model;
10
11 selectElement(uml::Namespace package, String name):
12   package.ownedMember.selectFirst(e|e.name==name);
13
14 createRequires(uml::NamedElement client, uml::NamedElement supplier) :
15   client.createUsage(supplier).setName(client.name+'_requires_'
16                                     +supplier.name);

```

Fig. 2. Excerpt of the model transformation for configuring the Lock Control Framework

In our case, when a specific authentication device is selected, the corresponding reader component is connected to the `LockControlMng` through the `IAccess` interface. In the same way, the `LockControlMng` component is connected to the corresponding authenticator component through the `IVerify` interface. Both the authenticator and the reader components must also be connected to `LockControlMng` through the `IRegister` interface. The components corresponding to non-selected alternatives must simply be removed. Similarly, the `DoorActuator` and `LockControl` components are adequately connected if the corresponding optional features are selected; otherwise, they should be removed.

Unlike *variability specification* and *variability realisation*, there are no well-known languages that are commonly used for specifying *product derivation* processes. With the emergence of Model-Driven techniques [5], it has been demonstrated [10,11] that product derivation processes can be unambiguously specified and automated using a model transformation language, such as ATL [6] or xTend [7]. The next section explains some of the shortcomings of specifying these rules using a common model transformation language and the motivation for engineering new languages that serve to unambiguously specify and automate product derivation processes for a software product line.

2.2 Product Derivation Process in a Common Model Transformation Language

Fig. 2 shows an excerpt of a model transformation in xTend for configuring the architectural model of the lock control framework contained in Fig. 1. This model transformation accepts as input two models (line 00): (i) the reference architecture model (`inputModel`, see Fig. 1 (bottom)), and (ii) a configuration of the feature model (`cfg`), i.e. a specific selection of variable features. Based on this selection, different subtransformation rules are invoked (lines 02-04).

These subtransformations add missing parts (e.g. connections between components) and remove other parts (e.g. pairs of unselected reader and authenticator components), according to the selected features. For instance, lines 06-09 show

the subtransformation for the case where an automatic lock is selected. Lines 07-09 add a requires relationship between the `LockControlMng` and the `ILockControl` interface, using two subfunctions that are `selectElement` and `createRequires`.

The main drawback of general-purpose transformation languages for specifying product derivation processes is that software product line engineers must have in-depth expertise in managing this kind of language. As the reader may notice, some of the terms used in this subtransformation (e.g. `uml::Namespace` or `uml::NamedElement`) are weakly related to the illustration contained in Fig. 1 (bottom). This is due to the following identified problems:

Metamodel Burden. A model transformation language is often based on abstract syntax manipulations. According to Jayaraman et al [12], “Most model developers do not have this knowledge. Therefore, it would be inadvisable to force them to use the abstract syntax of the models”.

Language Overload and Abstraction Mismatch. There are different kinds of model transformation languages [13], and each of them is based on a specific computing model. They range from rule-based languages (e.g. ATL [6]) to expression-based languages (e.g. xTend [7]) and graph-based languages (e.g. AGG [14]). When employing a model transformation language, software product line engineers must also understand the underlying computing style (e.g. rule-based) and learn the language syntax. As a result, software product line engineers are forced to rely on abstractions that might not be naturally part of the abstraction level at which they are working.

For instance, the subtransformation `createRequires` is invoked to add a “requires” relationship between a component and an interface. According to the UML metamodel, a “requires” relationship can be established between pairs of `NamedElements` (a metatype in the UML metamodel). Thus, this function accepts as parameter two named elements, invokes the `createUsage` function (a built-in function of xTend) with the other element as parameter (A “requires” relationship has a `Usage` metatype in the UML metamodel). Finally, the name of this relationship is set according to an in-house convention. As can be noticed, significant expertise in the UML metamodel is required. Moreover, the developer must also understand some details of the model transformation language, xTend in this case. For instance, the developer should know xTend is an expression-based language, where transformations are expressed as functions which compute expressions. Expressions can be concatenated using the “->” symbol. Some predefined functions, such as `createUsage`, can be used to facilitate the manipulation of UML models.

2.3 Solution Overview

This section explains our proposed solution for overcoming the problems discussed above. We propose to create dedicated languages for specifying product derivation processes at each stage in the software development lifecycle. These dedicated languages must:

```

// Variability description of the Control Lock SPL architecture
00 Concern LockControl {
01   VariationPoint AuthenticationDevice {
02     Kind: alternative ;
03     Variant Keypad {
04       SELECT:
05         connect(KeypadReader,LockControlMng) using interface(IAccess);
06         connect(KeypadReader,LockControlMng) using interface(IRegister);
07         connect(LockControlMng,KeypadAuth) using interface(IVerify);
08         connect(KeypadAuth,LockControlMng) using interface(IRegister);
09       UNSELECT:
10         remove(KeypadReader);
11         remove(KeypadAuth);}
12     ...
13   VariationPoint AutomaticLock {
14     Kind: optional;
15     SELECT:
16       connect(LockControl,LockControlMng) using interface(ILockControl);
17       connect(LockControl,LockControlMng) using interface(IRegister);
18     UNSELECT:
19       remove(LockControl);}
20     ....
21   }
// Configuration of a specific product
22 invoke(LockControl,AuthenticationDevice,Keypad);

```

Fig. 3. VML specification for the Lock Control Framework

1. Establish the link between the models that specify variability (*problem space*) and reusable software assets (*solution space*), specifying which actions must be carried out on software models (e.g. UML 2.0 models), when a certain decision is adopted on the problem space model (e.g. when an alternative is selected).
2. Provide a terminology familiar to the SPL engineer, using the abstractions and concepts commonly used at each software development stage.
3. Enable the automation of the product derivation process.

The *Variability Modelling Language* (VML) [15] is an example of this kind of language. VML can be used to specify product derivations of architectural models, such as depicted in Fig. 1. It is a textual language that facilitates composition of reusable architectural software assets and relies on typical abstractions of architectural designs, such as components or interfaces.

VML provides composition mechanisms to explicitly specify architectural variabilities relative to concerns of the SPL architects. The realisation of each architectural concern contains one or more variation points and variants. Composition primitives are defined to allow connecting, adding and removing elements, and so forth, in the architectural models.

Fig. 3 illustrates part of the VML specification for the lock control framework. A VML specification is composed of concerns (line 00). Each concern can contain several variation points, which can be alternative (line 01-02) or optional (line 13-21) variation points, among others¹. Alternative elements (e.g. Keypad, line 03)

¹ Readers interested in VML details can read our previous paper describing the VML constructs [15].

and optional elements (line 13) are composed of a sequence of actions that specify which operations must be carried out during the product derivation process if the variant is selected or unselected.

For instance, it is specified for the Keypad case (lines 03-12) that whenever a Keypad is selected (lines 04-08), the KeypadAuth and KeypadReader components must be connected to the LockControlMng component through the corresponding interfaces, as described in Section 2.1. The connect operator (lines 05-08) is an intuitive composition mechanism to specify that two components must be connected using the interface specified as a parameter. The first parameter of the connect operator is the component that requires the interface while the second parameter is the component that provides it. In the case where the Keypad variant is not selected (lines 09-12), the KeypadAuth and the KeypadReader components are removed from the architecture. The remove VML operator is used to remove architectural elements from the model.

VML also provides constructs for specifying the configuration of SPL products, i.e. selection of variants that must be included in a specific product. Variant selection is described in a VML configuration file, which contains a set of invoke operators. The operator refers explicitly to specific variation points, and their variants, within concerns. For example, specifying `invoke(LockControl, AuthenticationDevice, Keypad)` would select the Keypad variant that is part the Authentication-Device variation point, within the LockControl concern. The configuration file must conform to the VML syntax and the VML specification of the SPL. It must also satisfy the constraints between variants, such as for instance, that DoorOpener requires AutomaticLock (see Fig. 1) (top). Ensuring that a configuration satisfies this kind of constraints between features is beyond the scope of this paper.

VML fulfills the first two requirements of a dedicated language for specifying product derivation process: (1) it provides the link between feature models and architectural models; managing the complex “wiring” between decisions in the problem space (a feature model in this specific example) and their realisation in the solution space (a UML 2.0 model in this case); (2) it uses a terminology familiar to SPL software architects. For instance, VML provides a connect operator for connecting two components, instead of of something like that depicted in Fig. 2, lines 06-15.

It should be noticed that the general structure of VML can be applied to other software development stages, by simply providing a new set of operators to be placed in the SELECT and UNSELECT clauses. Hence, a VML for use cases could define an operator `extend(uc1,uc2) in point(ep1)`, which means that, as a consequence of selecting/unselecting a feature, the use case uc1 extends the uc2 using the extension point ep1.

Thus, VML-like languages, with a specific set of operators for each development stage, address the first two requirements of the three stated at the beginning of this section. In the following, we will denote a VML-like language, as VML4*, where “*” represents the model where variable software artefacts are modelled (e.g. UML architectural models, use cases). Automation of the product derivation process is the only remaining requirement for achieving our goal.

Automation can be achieved by transforming a VML4* specification into a set of model transformations expressed in a common model transformation language, such as ATL or xTend. The next section describes the main contribution of this paper, a process for creating “compilers” for VML4* languages to perform this transformation task. The process also covers the automatic generation of the tools for editing VML4* specifications.

3 Engineering a Language for Specifying Product Derivation Processes

This section describes how to implement compilers for VML4*, which automate SPL product derivation processes. It also explains how to automatically generate editors for VML4* languages.

3.1 Process Overview

Fig. 4 shows the process for engineering a VML4* language. This process is based on a suite of model transformations, which has been implemented in openArchitectureWare (oAW). Nevertheless, the concepts behind this process implementation do not specifically require oAW and they can be implemented using other technologies, such as the Epsilon suite [16].

Elements with a gray background in Fig. 4 are automatically generated. Elements with a white background and text in *italics* are elements that must be defined by the VML4* language implementers. Elements with a white background and normal text are elements that must be defined by the SPL engineers using the VML4* language. SPL engineers only need to know the syntax of the VML4* language to specify the product derivations. Each step of Fig. 4 is detailed in the following.

Language Syntax. First of all, the syntax of the VML4* language is defined using the EBNF notation (Fig. 4, label 1). Once the grammar has been defined, we

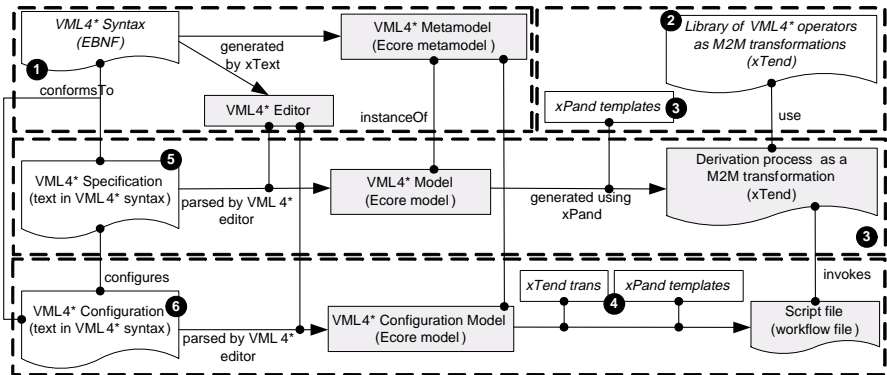


Fig. 4. Overview of the process for engineering a VML4* language

apply xText to the language grammar in order to generate the tooling required for editing and processing VML4* models.

Operator Implementation. Secondly, each operator of the language (e.g. connect) is implemented as a model to model transformation (Fig. 4, label 2). This model transformation has the same parameters as the language operator, and it provides the operationalisation for it. xTend, the model to model transformation language of oAW, is used for implementing these transformations.

High-order M2T Transformations. Next, a set of high-order model to text transformations are developed for converting VML4* specifications into textual model to model transformations, which implements the product derivation process (Fig. 4, label 3). Basically, these model to text transformations convert each action contained in a VML4* model into a textual invocation to the corresponding implementation of the VML4* operator. xPand, the model to text transformation language of oAW, is used for implementing these transformations.

Product Derivation. Then, model transformations must be developed for executing the product derivation processes (Fig. 4, label 4). Such model transformations accept as input a VML4* configuration file, i.e. a valid configuration of variants, and they generate a script file that executes the product derivation process. This script file is in charge of invoking the generated model transformations that implement the product derivation process with the *solution space* model (e.g. a UML architectural model) as input. The model of a specific product is obtained as a result.

The steps described until now are the steps typically carried out for implementing a VML4* language. Using this tooling, SPL engineers can specify product derivation processes and execute them as follows:

1. First, SPL engineers create a VML4* description of a SPL (Fig. 4, label 5). This textual VML4* specification is converted into a model instance of the VML4* metamodel by a parser generated by xText. Then, VML4* is “compiled” using the model to text transformations. As a result, a product derivation process implemented as a set of model-to-model transformations is obtained.
2. A configuration, i.e. a valid selection of variants, for the previous VML4* description is created (Fig. 4, label 6). This configuration is “executed” using model transformations, and a script file is obtained and executed. As a result, the model of a specific product is obtained (e.g. an architectural model).

The result of the steps above is that SPL models and product designers remain unaware of intricate model transformation details. VML4* offers terminology familiar to SPL engineers, working at a certain abstraction level, for specifying product derivation processes. Then, using this specification, a product derivation process in a common model transformation language (i.e. xTend) is automatically generated. Details of the model transformation language (e.g. xTend) and the metamodel of the software model (e.g. the UML metamodel) are hidden from SPL engineers. The next sections show each one of the elements of this process more in detail.


```

00 Concern: "Concern" name = ID "{" (variationPoints+=VariationPoint)* "}";
01 VariationPoint: "VariationPoint" name = ID "{" "Kind:" kind=Kind ","
02   (variants+=Variant)* (nonVariants=NonVariant)? "}";
03 Variant: "Variant" name=ID "{"
04   "SELECT:" (selectedActions+=Action)*
05   "UNSELECT:" (unselectedActions+=Action)* "}";
06   ....
07
08 Action: SimpleConnect | Remove | Merge |Set;
09 SimpleConnect :
10   "connect" "(" (" requiredComponentName = ID "," providedComponentName = ID ") " using"
11   "interface" "(" (" interfaceName = ID ") " ";";
12 Remove: "remove" "(" ("componentName = ID") " ";";

```

Fig. 5. Defining the grammar for VML4Architecture

3.2 Automatic Generation of the Language Editor

The first step in the engineering process of a VML4* language is to define the grammar of the language. Fig. 5 shows part of the grammar for a version of VML for architectural models. This grammar is defined using a notation used by the xText tool, which is similar to EBNF. Most keywords and the skeleton of the grammar can be reused across different VML4* languages. For instance, all the rules for defining the structure of *VariationPoints* and *Variants* (Fig. 5, lines 00-06) can be used for all VML4* languages. Basically, what an engineer defining a new VML4* language only needs to do is to define the new set of operators that will be placed in the *SELECT* and *UNSELECT* clauses for a VML4* language. Fig. 5, lines 08-12, illustrates the definition of the *connect* and *remove* operators of the VML for architectural models shown in Fig. 3.

Once the grammar has been defined, we apply xText to the language grammar. xText is a framework for facilitating the development of textual domain specific languages (DSLs). It accepts as input the grammar of a language, and creates a parser, an Ecore metamodel and a textual editor (as a Eclipse plugin, with syntax highlighting and error messages) for that language. Fig. 6 shows a screenshot of the automatically generated editor of of the VML for architectural models. Thus, the application of xText to the language syntax generates the tooling required for writing VML4* specifications in Eclipse and obtaining models of these specifications, the models being instances or conforming to the language metamodel.

3.3 Implementation of the of Operators

Each VML4* operator has a semantics that can be defined in terms of “drawing or removing lines and boxes” from a model, i.e. as a set of manipulations on the model that can be implemented as a model transformation. Thus, the idea is to implement each operator of a VML4* as a model transformation.

For instance, in the case of the *connect(req,prov) using interface(l)* operator, (See Fig.3), this operator connects the component *req* with the component *prov* through the interface *l*. The component *req* requires the interface, and the component *prov* provides it. The semantics of this operator can be described

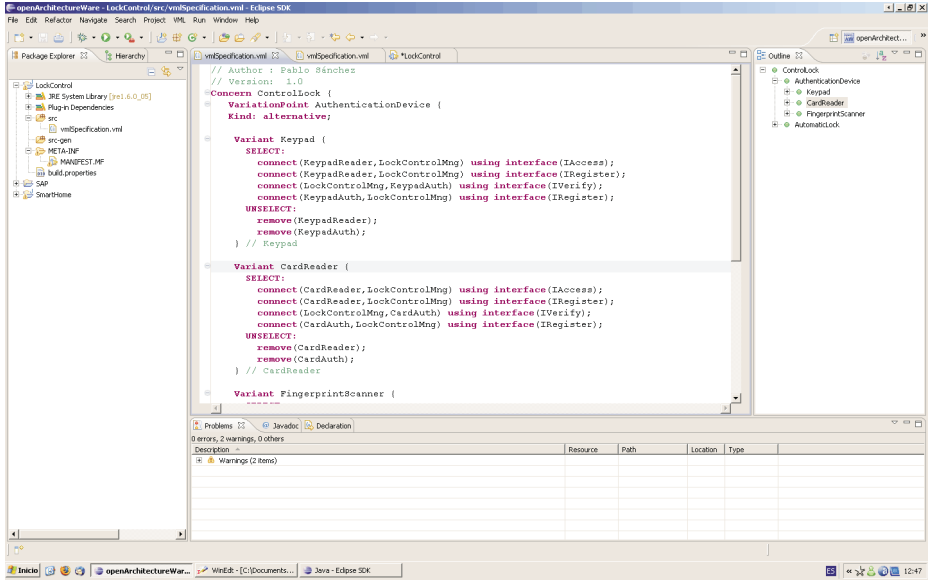


Fig. 6. Generated editor for VML4Architecture

informally as follows (we suppose *req*, *prov* and *I* exists, and the connections can be performed):

1. If component A has no required relationship with interface *I*, add this require relationship. Otherwise, do nothing.
2. If component B has no provided relationship with interface *I*, add this require relationship. Otherwise, do nothing.

This algorithm is implemented by the model transformation depicted in Fig. 7. Line 00 is the header of the model transformation that implements the *connect* operator. This transformation makes use of the helper functions *provides*, *addProvides*, *requires* and *addRequires*. The last two are shown in Lines 08-09 and Lines 11-12.

```

00 connect(uml::Model model, String reqComp, String provComp,String interface):
01   let req = model.selectComp(reqComp) :
02   (let prov = model.selectComp(provComp) :
03     (let int = model.selectInterface(interface) :(
04       prov.provides(int)?null:prov.addProvides(int)->
05       req.requires(int)?null:req.addRequires(int)->model))) ->
06   model;
07
08 addRequires(uml::NamedElement client, uml::NamedElement supplier) :
09   client.createUsage(supplier).setName(client.name+'_requires_'+supplier.name);
10
11 requires(uml::Component comp, uml::Interface int):
12   comp.required.contains(int);

```

Fig. 7. Implementation of the connect operator

These functions require some expertise in model-driven techniques and the UML metamodel. For instance, the parameters of the `addRequires` function are of type `uml::NamedElement`, which is a super(meta)type of `uml::Component` and `uml::Interface` and many other UML elements, such as classes. Using `uml::NamedElement` as type for the input parameters, the `addRequires` function can be reused for creating requires relationships between other kinds of UML elements, such as between a class and an interface. These picky details of the UML metamodel are hidden from the SPL engineer using a VML4* language. SPL engineers do not need to be aware of them.

3.4 Compiling the Language

Once the operators of a VML4* language have been implemented, the model-to-text transformations for converting a VML4* specification into a executable model-to-model transformation are developed. These high-order model-to-text transformation accept as input a VML4* specification and execute the following steps:

1. A VML4* model is composed of several concerns (Section 2.3). For each concern, the transformations generate a file that will contain the model transformation for configuring that concern. For instance, for the `LockControl` concern (see Fig. 3), the `LockControl.ext` file, containing a model-to-model transformation in xTend is generated.
2. A concern is composed of a collection of variation points. We iterate over this collection, processing each variation point. A variation point can be either an *optional* or an *alternative* variation point. The latter is further composed of a collection of variants. Each variant and optional variation point is processed.
3. Variants and optional variation points are composed of two different set of operators: (1) the corresponding ones to the `SELECT` clause; and (2) the corresponding ones to the `UNSELECT` clause. Thus, for each variation `var`, two model transformations are created, one for the case the variation is selected (`select%var%`) and one for the case the variation is unselected (`unselect%var%`).
4. For each one of these `select%var%/unselect%var%` transformations, the corresponding actions are processed. Each action in a `SELECT/UNSELECT` clause is converted to a call with specific values, in xTend, to a VML4* operator of the library presented in the previous section.

It should be noticed that the first three steps are reusable for different VML4* languages, since these steps are not dependent of the operators used for each VML4* language. The model-to-text transformation for the fourth step must be reengineered for each VML4* language, since it depends on the set of operators selected.

Fig. 8 shows an excerpt of this model transformation implemented as a xPand template, which processes the version of VML for architectural models. This transformation accepts as input (Line 00) a VML model of a SPL (called `ConcernFile` in the VML metamodel). Then, for each concern contained in this `ConcernFile`, a file named `%concernName%.ext` is created. Then, each variation point contained

```

00 <<DEFINE co-pile FOR ConcernFile>>
01   <<FOREACH concerns AS c/nc>>
02     <<FILE conc.name+".ext">>
03   extension net::ample::wp2::vml::library::umlU4il;
04   extension net::ample::wp2::vml::library::vmlOperators;
05   <<FOREACH conc.variationPoints AS variation>>
06     <<IF (variation.kind == Kind::Alternative)>>
07       <<FOREACH variation.variants AS variant>>
08   select"variant.name"(uml::Model model):
09     <<EXPAND processAction FOREACH variant.selectedActions>>
10     model;
11
12   unselect"variant.name"(uml::Model model):
13     <<EXPAND processAction FOREACH variant.unselectedActions>>
14     model;
15     <<ENDFOREACH>>
16     <<ELSEIF (variation.kind == Kind::Option)
17   ...
18 <<ENDIF>><<ENDFOREACH>><<ENDFILE>><<ENDFOREACH>><<ENDDEFINE>>
19
20 <<DEFINE processAction FOR Action>>
21   <<IF (this.metaType == vml::SimpleConnect)>>
22     <<EXPAND connect FOR (vml::SimpleConnect) this>>
23   <<ELSEIF (this.metaType == vml::Remove)>>
24     ...
25   <<ENDIF>>
26 <<ENDDEFINE>>
27
28 <<DEFINE connect FOR SimpleConnect>>
29   model.connect('this.firstComponentName', 'this.secondComponentName',
30     'this.interfaceName')-><<ENDDEFINE>>
31 <<DEFINE remove FOR Remove>>
32   model.remove('this.componentName')-><<ENDDEFINE>>

```

Fig. 8. Excerpt of the templates for processing VML

in the concern is processed (Line 05). Each variant is processed differently depending on if it is an alternative (Line 06), an option (Line 16) or other kind of variation. In the alternative case, for each variant, the model transformation for the select (Lines 08-10) and unselect case (Lines 12-14) are executed. These transformations creates the header for the model transformation and processes all the corresponding actions (the selected and the unselected ones) using the `processAction` subtemplate. This subtemplate (Lines 20-26) calls to a specific subtemplate depending on the metatype of each action. Each specific subtemplate generates the corresponding invocations to the implementations of the VML operators, with the corresponding values. For instance, the `remove` operator generates the `xTend` expression of Line 32, which invokes the implementation of the `remove` operator.

```

00 selectKeypad(uml::Model model):
01   model.connect('KeypadReader', 'LockControlMng', 'IAccess')->
02   model.connect('KeypadReader', 'LockControlMng', 'IRegister')->
03   model.connect('KeypadAuth', 'LockControlMng', 'IRegister')->
04   model.connect('LockControlMng', 'KeypadAuth', 'IVerify')->model;
05
06 unselectKeypad(uml::Model model):
07   model.remove('KeypadReader')->
08   model.remove('KeypadAuth')->model;

```

Fig. 9. Excerpt of the templates for processing VML

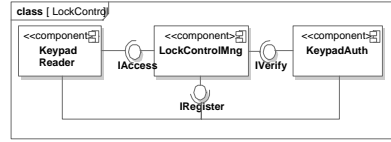


Fig. 10. Architecture for a specific product automatically generated

Fig. 9 shows part of the result of applying these templates to the VML specification of Fig. 3. As an output of “compiling” the VML specification, two model transformation functions are generated for the **Keypad** variant (see Fig. 3), corresponding to its **SELECT** and **UNSELECT** clauses. These functions automate the actions to be carried out when a **Keypad** is either selected (lines 00-04) or unselected (lines 06-08). Once again, the SPL engineer is completely unaware of these xPand templates and the details of this transformation process.

Once the implementation in a model transformation of a product derivation process has been obtained, we need only to call these generated functions according to the selection of variants in order to automatically obtain the model of a specific product. This is explained in the next section.

3.5 Generation of Specific Products

Finally, in order to obtain a specific software product, SPL engineers select a set of variants according to the end-user preferences. This set of variants is expressed as a VML configuration file (see Fig. 3, line 22), which contains a set of **invoke** statements that specify the set of selected variants. This configuration file is processed as follows:

1. The configuration file indicates only those variants that have been selected. We apply first a model transformation, using the configuration model as input, which returns an auxiliary model with all the unselected variants.
2. Then, we run a model to text transformation that generates a script (in the oAW scripting language), which, when executed, generates the architecture of the specific product with the selected variants. For each selected variant **Var**, the script statement for invoking the **select%Var%** model to model transformation is generated. Similarly, using the auxiliary model with the unselected variants, we generate statements in the scripting language that invoke the **unselect%Var%** transformation corresponding to each selected variant.

Finally, this script file is executed using the *solution space* model as input. The execution of the invoked transformations carries out the required modifications on the software model for selecting and unselecting variants. As a result, the software model of a specific product is obtained. It should be noticed that these transformations are independent of the VML4* specific operators. Hence, they can be reused in any VML4* language.

In the case of the VML4Architecture specification of Fig. 1), the architecture model for a specific product is obtained, using the configuration file depicted. The

architectural model of Fig. 10 would be obtained automatically. In this model, the `KeypadAuth` and `KeypadReader` components are appropriately connected to the `LockControlMng`; the `DoorActuator` and the `LockControl` components plus the `lDoor` and `lLockControl` interfaces removed.

As before, the SPL engineer must also take care of creating the configuration file, i.e. a set of `invoke` statements that corresponds to the customer requirements, but he or she is unaware of details of the transformations carried out for the automatic generation of software models of specific products.

4 Discussion and Related Work

This section provides a reflective analysis of our proposal by contrasting it with related work.

There are several commercial tools, such as `pure::variants` [3] and `Gears` [4], which target the automation of product SPL derivation processes. However, they focus on the generation of implementation artifacts, deriving code for a specific product from family or template implementations. These tools would need to be significantly extended for dealing with models, although such extensions would be not trivial as they are often based on XMI-based manipulations.

Ziadi et al [10] were among the first authors to propose the implementation of product derivation processes as model transformations. Their proposal relies on the realisation of product derivations via a model transformation language, namely MTL. Botterweck et al [11] propose the same idea, but they use ATL as model transformation language. As discussed in Section 2.2, this strategy requires that SPL engineers deal with low-level details of model transformation languages. Our approach provides a process for engineering languages that use syntax and abstractions familiar to the SPL engineers. This eliminates the burden of understanding intricacies associated with model transformation languages commented in Section 3.3. From VML4* specifications, an implementation of the product derivation process in a model transformation language is automatically generated, but SPL engineers are unaware of this generation process, being unaware of intricacies of model transformation languages.

Oldevik and Haugen [17] propose using high-order transformations for simplifying the model transformation rules that implement a product derivation process. In order to achieve this goal, they extend an existing model-to-text transformation language called MOF-Script with aspects. These aspects are managed as high-order transformations that accept as input a model transformation and generate as output a modified model transformation. Base assets of the SPL are generated by a base model transformation, which is enhanced by MOFScript aspects implementing the different variants. Nevertheless, although high-order transformations can help to simplify the development of model transformations that implement a product-derivation process, they are still model transformations, having the same problems as those identified in Section 2.2 and discussed in the previous paragraph. Furthermore, the incorporation of aspects to a model

transformation language increases the number of constructs and keywords of the language, increasing the *language overload* problem.

There are a wide number of approaches for software product line engineering that are not based on the use of feature models for specifying variability inherent to a family of products. They are mainly based on the creation of some kind of Domain-Specific Modelling Languages (DSMLs), which capture the variability of a family of products [18]. Different products are instantiated by means of creating different models that conform to the metamodel of the DSML. A specific model serves as input for a set of model transformations which generates the specific product. However, engineers must still deal with model transformation languages.

The reader, who is knowledgeable, in the area could claim DSMLs are more expressive than feature models, since there are certain kinds of variabilities, such as structural variability (e.g. a SmartHome having a different number of floors and rooms), which can not be expressed using only alternative and optional features. Nevertheless, a recent technical report [19], demonstrate there are a correspondence between cardinality-based feature models and metamodels, and both can be used to express the variability of a software product line.

FeatureMapper [20] is a tool that allows to associate features with arbitrary elements of an Ecore-based model. Nevertheless, FeatureMapper does nothing beyond establishing the wiring between *problem space* and *solution space*, i.e. FeatureMapper does not deal with product derivation processes.

Czarnecki and Antkiewicz [21] present an approach similar to ours. They create a *template model*, which plays the role of the model for all the family of products. Variability is modelled using feature models. Elements of this model are annotated with *presence conditions*. These presence conditions are written using special actions, comparable to VML4* operators, called meta expressions. Given a specific configuration, the presence conditions gives as a result true or false. If the result is false, they are removed from the model. Thus, such a template-based approach is specific to negative variability, which might be critical when a large number of variations affect a single diagram. VML4* operators can also support positive variability by means of operators such as *connect* or *merge* operators. We defined a *merge* operator, based on the UML merge relationship, for a VML4Architecture language [15]. This operator supports the definition of variants in separate models that are later combined by this operator. Moreover, unlike to VML4*, presence conditions imply introducing annotations to the SPL model elements following an invasive approach. Therefore, the actions associated with a feature selection are scattered across the model, which could also lead to scalability problems. In our approach, they are well-encapsulated in a VML4* specification, where each variant specifies the actions to be executed.

MATA [12] and XWeave [8] are two aspect-oriented modelling tools that have been applied in a SPL for feature composition. In both cases, each variant is modelled as an aspect in a separate model. An aspect in these tools is similar to a graph-based transformation. The pointcut plays the role of the LHS (Left-Hand Side) pattern, and the advice the role of the RHS (Right-Hand Side)

pattern. When the LHS pattern matches the primary or core model, it is substituted by the RHS pattern. Both approaches have difficulties when managing small variants, i.e. variant models with a reduced number of elements. Let us suppose we have an interface with 10 operations, 8 of them variable and corresponding to different variants. Then, we would need to create 8 MATA or XWeave aspects, each one corresponding to one variant and containing a particular operation. This large number of small aspects would lead to scalability and maintenance problems as the number of variants grows. VML4* languages support both fine-grained and coarse-grained variant management. Small variants are managed through positive (e.g. `connect`) and negative (e.g. `remove`) operators. Coarse-grained variability can be supported by means of a `merge`-like operator [15], based on the UML merge, which serves to compose different model fragments, similarly to MATA and XWeave.

Hendrickson and van der Hoek [22] present an approach for modelling SPL architectures using change sets. Similarly to MATA and XWeave, a variant is modelled as a change set on a core model. This change set can add elements or remove and modify elements of the core model. This approach has the same problems regarding fine-grained variability as XWeave and MATA.

Summarising, our approach, compared to other approaches, provides better scalability and maintainability of *solution space* SPL models by:

1. Supporting the specification of SPL product derivation process, or *variability composition*, using a syntax and a set of abstractions familiar to SPL engineers, avoiding them having to deal with low-level intricacies of model transformation languages. Moreover, automation of the SPL product derivation process is preserved by transforming VML4* specifications into a set of model transformations specified in a common model transformation process. Details of this transformation process are hidden from the SPL engineers.
2. Separating actions associated with variants from SPL models, instead of writing actions directly over SPL models, such as Czarnecki and Antkiewicz [21].
3. Providing operators for positive and negative variability.
4. Systematically supporting both small and large variants. Variants can be modelled in separate models or not, as desired. Moreover, each variant only contains those actions that serve to manage the variant itself. Thus, the addition of new variants, or the removal of existing ones, does not affect other variants, contributing to maintenance and evolution.

Operators defined for a specific VML4* language are dependent on that target language. The syntax of some operators, can be applied to practically any model at the abstraction level. For instance, `connect` and `remove` are operators that can be applied to practically all architectural modelling languages. Unfortunately, since model transformations depend on a specific metamodel, the implementation of these operators must change if the target language changes.

Finally, we would like to point out that although the examples presented in this paper focus on structural architectural models, VML4* can be also applied to models describing behavioural views of a system. We refer the interested reader to [15,23].

5 Conclusions and Future Work

This paper has presented an innovative process that allows the engineering of languages, called VML4*, for specifying product derivation processes of SPL, using a syntax and a set of abstractions familiar to SPL engineers. We have also shown how a set of model transformations that automate the product derivation process is automatically generated from the VML4* specifications. This process has been illustrated engineering a VML4Architecture, where the target language is UML 2.0, which is used for the construction of architectural models. We have used the VML4Architecture for specifying the product derivation process of a control lock framework. In addition to the control lock framework presented in this paper, we have also applied the VML4Architecture to a persistence manager, a multi-agent product line, and a public health complaint framework [15]. In addition to VML4Architecture, we are constructing, as part of the AMPLE² project, a VML4Requirements, where the VML operators manage UML use case models and activity diagrams capturing requirements [23]. As future work, we will continue creating more VML4* for different target languages. It is also our intention to explore how the process described in this paper can also be applied to programming languages, using *program transformation* languages, such as Stratego [24], instead of model transformation languages.

References

1. Pohl, K., Bockle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005)
2. Käkölä, T., Dueñas, J.C.: Software Product Lines: Research Issues in Engineering and Management. Springer, Heidelberg (2006)
3. Beuche, D.: Variant Management with pure:variants. Technical report, pure-systems GmbH (2003)
4. Krueger, C.W.: BigLever Software Gears and the 3-tiered SPL Methodology. In: Companion to the 22nd Int. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Montreal, Quebec, Canada, pp. 844–845 (October 2007)
5. Beydeda, S., Book, M., Gruhn, V.: Model-Driven Development. Springer, Heidelberg (2005)
6. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
7. (April 2008), <http://www.openarchitectureware.org/>
8. Völter, M., Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: Proc. of the 11th Int. Software Product Line Conference (SPLC), Kyoto, Japan, pp. 233–242 (September 2007)
9. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Staged Configuration through Specialization and Multilevel Configuration of Feature Models. Software Process: Improvement and Practice 10(2), 143–169 (2005)
10. Ziadi, T., Jézéquel, J.M.: Software Product Line Engineering with the UML: Deriving Products. In: [2], pp. 557–588

² AMPLE project: <http://www.ample-project.net>

11. Botterweck, G., O'Brien, L., Thiel, S.: Model-Driven Derivation of Product Architectures. In: Proc. of the 22nd Int. Conference on Automated Software Engineering (ASE), Atlanta, Georgia, USA, pp. 469–472 (2007)
12. Jayaraman, P., Whittle, J., Elkhodary, A.M., Gomaa, H.: Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 151–165. Springer, Heidelberg (2007)
13. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–645 (2006)
14. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 446–453. Springer, Heidelberg (2004)
15. Loughran, N., Sánchez, P., García, A., Fuentes, L.: Language Support for Managing Variability in Architectural Models. In: Pautasso, C., Tanter, É. (eds.) SC 2008. LNCS, vol. 4954, pp. 36–51. Springer, Heidelberg (2008)
16. Kolovos, D.S., Paige, R.F., Polack, F.A.: Novel Features in Languages of the Epsilon Model Management Platform. In: Proc. of the Int. Workshop on Models in Software Engineering (MiSE), 30th Int. Conference on software Engineering (ICSE), Leipzig (Germany), pp. 69–73 (May 2008)
17. Oldevik, J., Haugen, Ø.: Higher-Order Transformations for Product Lines. In: Proc. of the 11th Int. Conference on Software Product Lines (SPLC), Kyoto, Japan, pp. 243–254 (September 2007)
18. Tolvanen, J.-P., Kelly, S.: Defining domain-specific modeling languages to automate product derivation: Collected experiences. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 198–209. Springer, Heidelberg (2005)
19. Stephan, M., Antkiewicz, M.: Ecore.fmp: A tool for editing and instantiating class models as feature models. Technical Report Tech. Rep. 2008-08, ECE, University of Waterloo (May 2008)
20. Heidenreich, F., Kopcsek, J., Wende, C.: Featuremapper: mapping features to models. In: Companion of the 30th International Conference on Software Engineering (ICSE), Leipzig, Germany, pp. 943–944 (May 2008)
21. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005)
22. Hendrickson, S.A., van der Hoek, A.: Modeling Product Line Architectures through Change Sets and Relationships. In: Proc. of the 29th Int. Conference on Software Engineering (ICSE), Minneapolis, Minnesota, USA, pp. 189–198 (May 2007)
23. Alférez, M., Kulesza, U., Sousa, A., Santos, J., Moreira, A., Araújo, J., Amaral, V.: A Model-driven Approach for Software Product Lines Requirements Engineering. In: Proc. of the 20th Int. Conference on Software Engineering and Knowledge Engineering (SEKE), San Francisco Bay, California, USA (July 2008)
24. 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)

Transformation Language Integration Based on Profiles and Higher Order Transformations*

Pieter Van Gorp, Anne Keller, and Dirk Janssens

University of Antwerp

{pieter.vangorp, anne.keller, dirk.janssens}@ua.ac.be

Abstract. For about two decades, researchers have been constructing tools for applying graph transformations on large model transformation case studies. Instead of incrementally extending a common core, these competitive tool builders have repeatedly reconstructed mechanisms that were already supported by other tools. Not only has this been counter-productive, it has also prevented the definition of new language constructs independently of a specific transformation tool. Moreover, it has complicated the comparison of transformation languages. This paper describes a light-weight solution to this integration problem. The approach is based on executable transformation modeling using a small UML profile and on higher order transformations. It enables the integration of graph transformation tools such as Fujaba, VMTS and GReAT. The paper illustrates the approach by discussing the contribution of a Copy operator to any of these tools. Other language constructs can be realized similarly, without locking into specific tools.

1 Problem: Lack of Portability and Reuse

This paper tackles the integration problem of model transformation tools in general but focuses on graph transformation tools. Graph transformation gained industrial credibility in the nineties, thanks to the application of the Progres language (and tool) within industrial tool integration projects [1]. One of Progres' shortcomings was that it relied on a rather limited language for metamodeling. Moreover, some of its control flow constructs were rather ad-hoc compared to standard notations such as activity diagrams. After working within the Progres team, Zündorf showed how these two limitations could be overcome by defining "Story Diagrams" as a new graph transformation language based on the UML [2]. Due to some interoperability issues with the C implementation of the Progres tool [3], a new Java based tool called Fujaba was implemented for these Story Diagrams [4]. Unfortunately, no mechanisms for reusing the advanced pattern matching libraries of Progres were available. Consequently, a lot of existing functionality had to be implemented again. In fact, some of Progres' language constructs are not yet supported by any other tool. In parallel to Fujaba, another graph transformation tool called GReAT was implemented for the

* This work was funded by the SBO grant (IWT-60831) of the Flemish Institute supporting Scientific-Technological Research in industry (IWT).

Windows platform [5]. Similar to Fujaba and Progres, GReAT also supported pattern matching, rewriting, rule iteration, etc. However, one did not yet define metamodels for integrating the different tools' editors, analyzers or compilers.

Although this kind of prototyping was natural for the first generation of tools, the disadvantages became clearly visible when even more tools were constructed. With the advent of the MDA, a new generation of tools (such as MOLA [6] and VMTS [7]) was constructed by competing research groups. Unfortunately, these tools have proposed yet another syntax for graph transformation constructs with the same underlying semantics. Moreover, none of the graph transformation tools rely on libraries to reuse existing infrastructure for pattern matching, control flow, etc. Although the existence of multiple tools ensures healthy competition, today's tool landscape suffers from (1) the lack of portability of transformation models across different editors, and (2) the lack of reuse of compiler/interpreter infrastructure. This paper presents language engineering techniques to solve these two problems.

The remainder of the text is structured as follows: Section 2 explains how standard UML profile syntax enables editor and repository independence, Section 3 describes how transformation language "behavior" can be shared across tools, Section 4 describes how the approach has been applied to contribute a new transformation language construct in a tool-independent manner, Section 5 discusses how the approach still applies for quite advanced and seemingly tool-specific language constructs, Section 6 presents related work, Section 7 presents an outlook for future work and Section 8 concludes.

2 Extensible Transformation Language: Standard Profile

The first step to transformation tool integration is the agreement on a metamodel for all mainstream transformation language constructs. The metamodel should meet the concerns of transformation writers, while these may be in conflict with those of transformation tool builders. Since transformation writers need to write and debug transformation models, they want a transformation language with a human-readable concrete syntax. Moreover, they want to be able to use the editor of the most user-friendly transformation tool (automatic completion, interactive debugging, ...). Finally, they may want to rely on the optimizer of one tool and potentially deploy the result using integration components of yet another transformation framework (EMF compatible Java, JMI compatible Java, Windows compatible C++, Unix compatible C++, ...). Since transformation tool builders need to provide such functionality, they want a simple metamodel and reuse existing software whenever possible.

2.1 Metamodel: UML 1.5

It turns out that a small subset of the UML 1.5 metamodel reconciles the requirements of transformation writers with those from tool builders. More specifically, UML class and activity diagrams resemble the concrete syntax of popular

graph transformation languages such as Story Diagrams, MOLA and VMTS quite closely.

UML 1.5 relates to an ISO standard and is supported by quite a number of mature tools (commercial as well as open source). In contrast, UML 2 based tools tend to differ more at the metamodel level. Moreover, the UML 2 metamodel is designed to integrate several diagram types that are irrelevant for transformation modeling. This complicates the metamodel structure and thus unnecessarily increases the implementation effort for tool builders. Transformation writers that insist on using a UML 2 editor can apply a standard converter for mapping UML 2 profile, class and activity models back to UML 1.5.

At the other end of the *universal-to-domain-specific* spectrum, the Graph Transformation eXchange Language (GTXL [8]) does not suffer from the metamodel complexity overhead associated with the UML standards. However, besides more problems discussed in Section 6, GTXL models have no standard *concrete syntax* beyond XML. Therefore, it does not satisfy the readability concerns of transformation writers. Section 6 also discusses the drawbacks of the Queries/Views/Transformations (QVT [9]) languages. Among other problems, QVT standardizes even more than is supported by mainstream transformation tools. Therefore, it requires much more implementation effort than the lightweight UML approach presented here.

2.2 Extensions for Transformation Modeling

Among the domain-specific extensions, a UML based transformation modeling language needs a means to relate an activity in a control flow to a particular rewrite rule. Such extensions could be realized using MOF extensions of the UML metamodel. However, this approach is undesirable for several reasons. First of all, such heavy-weight extensions break the compatibility of the standard transformation language with general purpose UML editors. More specifically, at the implementation level there would be several incompatibilities in MOF repository interfaces, XMI schema's, etc. Similarly, transformation tools that would use MOF extensions to realize language constructs beyond the core transformation standard would break compatibility with tools that only support that core.

Fortunately, the UML metamodel is designed for enabling language extensions without the need for a metamodel change. Essentially, there is a mechanism to introduce (1) the equivalent of a new metaclass (MOF M2) as a library element (MOF M1), and (2) the equivalent of a new MOF attribute (MOF M2) as another kind of library element (MOF M1). The mechanism is known as “UML Profiles”. New *virtual* metaclasses are called “Stereotypes” and new *virtual* meta-attributes are called “Tagged Values”.

Table 1 displays the concrete stereotypes and tag definitions from the proposed UML Profile for Transformation Modeling. The *core* profile only supports the basic graph transformation concepts: it has the notion of a rewrite rule (matched elements, created elements, deleted elements and updated elements) and control flows (iterative loops, conditionals and called transformations).

Table 1. A basic (*core*) UML Profile for Transformation Modeling

Profile Element	Related UML Constructs	Meaning
«ModelTransformation»,	Method	Callable Transformation
<i>motmot.transformation</i> tag	Method, Package, Activity Diagram	Link from method to controlled rewrite rules
«loop», «each time»	State Transition	Iterative loop over rule, iterative execution of nested flow
«success», «failure»	Transition	Match/Mismatch of rule
<i>motmot.constraint</i> tag	State, String Class, String	Application condition on state Application condition on node
«link», «code»	State, String	Call to transformation
<i>motmot.transprimitive</i> tag	State Package, Class Diagram(s)	Link from state in a flow to a rewrite rule
<i>motmot.metatype</i> tag,	Class	Type of rewrite node
«bound»,	Class	Parameter node, or node matched by previously executed rule
«create», «destroy»	Class, Association	Created/Destroyed elements
	Attribute (initial value)	Node attribute updates
«closure»,	Association	Transitive closure for link label

These syntactical constructs already provide a large portion of the expressiveness of today's graph transformation languages. Nevertheless, we do not want to standardize all transformation tools prematurely to this common denominator. Instead, new stereotypes and tagged values can be defined in UML models that extend the core profile. Such extensions can be made publicly available to extend any other UML editor with the new transformation language constructs. In summary, the proposed standard metamodel for exchanging transformation models consists of the class diagram (*core*), activity diagram (*activities*), and profile (*extension mechanisms*) packages of the UML metamodel.

2.3 Evaluation of the Core Transformation Modeling Profile

To validate whether the core transformation profile is still human-friendly, it has been used to model a variety of transformation problems using off-the-shelf UML tools. More specifically, several refactorings (*Pull Up Method*, *Encapsulate Field*, *Flatten Hierarchical State Machine* and others) and refinements (e.g., a translation of UML to CSP) have been modeled using commercial tools, without any plugins specific to the proposed transformation modeling profile [10]. A cognitive evaluation based on the framework of Green [11] confirms that the standard transformation language is usable even without any user interface implementation effort from transformation tool builders. Obviously, in an industrial context, one would rely on auto-completion and debugging plugins. However, the effort required for constructing such plugins would still be smaller than the effort for constructing a QVT editor from scratch.

As discussed above, the proposed integration architecture enables transformation tool builders to specialize in complementary features. One tool builder

(e.g., the MOLA team) can provide syntactical sugar on top of the proposed profile while another one (e.g., the VMTS team) can invest in optimizations. One tool builder (e.g., the MOFLON team) can provide integration with MOF based repositories (EMF and JMI) while another tool builder (e.g., the Progres team) can offer a C++ backend. To evaluate this architecture in practice, the MoTMoT tool provides a bridge from the proposed profile to JMI based repositories without offering a dedicated transformation model editor. Instead, MoTMoT relies on third-party editors such as MagicDraw 9 or Poseidon 2 [12].

To illustrate the core profile, Figure 1 shows the control flow and a rewrite rule for the *Extract Interface Where Possible* refactoring. Within the context of a package, the refactoring should mine for commonality between classes. More specifically, by means of an interface it should make explicit which classes implement the same method signatures. The transformation presented in this paper has been created by master-level students, after a three hour introduction to the proposed approach to transformation modeling.

The transformation flow shown on Figure 1 (a) controls the execution of three rewrite rules. The *matchOperations* rule associated with the first activity is used to check the precondition of the refactoring. More specifically, it matches only for those methods that have the same signature. Since the activity is decorated with the «loop» stereotype, it is executed for every possible match. Moreover, the two subsequent activities are executed for each match as well due to the «each time» stereotype on the outgoing transition. Due to the absence of a «loop» stereotype, the second activity is executed only once for each match of the first activity. The second activity is associated with the rewrite rule shown in Figure 1 (b). The *class1*, *class2*, *m1* and *m2* nodes are bound from the execution of the previous rule. The rule matches the visibility of the second method (*m2*) in order to create a copy of that method in the node *commonOperation*. The operation is embedded in the *newInterface* node that is newly created, as specified by the «create» stereotype. The newly created *interfaceLink1* and *interfaceLink2* elements ensure that *class1* and *class2* implement the new interface.

The third activity is executed for all possible matches of its rewrite rule. In this rewrite rule (not shown due to space considerations), all parameters from the new operation are created based on the parameters of *m1* and *m2*. To complete the discussion of this example, consider the two final states shown in the top of Figure 1 (a). The left final state is reached when the precondition (activity 1) fails. The right final state is reached otherwise. The final states are used to return a success boolean value for the transformation that is modeled by Figure 1 (a). Specific tools provide small variations on the syntactical constructs presented in this example. For example, the Fujaba and MOLA tools visually embed the rewrite rules (cfr., Figure 1 (b)) inside the activities of the control flow (cfr., Figure 1 (a)). However, for example VMTS does not rely on such an embedded representation. Therefore, the proposed concrete syntax resembles mainstream graph transformation languages in a satisfactory manner.

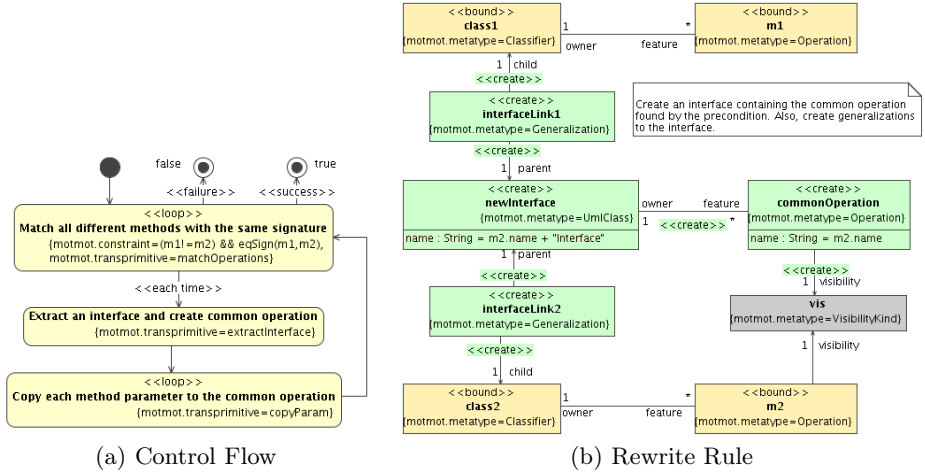


Fig. 1. *Extract Interface* refactoring, modeled using the core profile

3 Exchangeable Semantics: Higher Order Transformations

In the previous section we showed how transformation languages can be integrated at the syntactical level. This relates to the editors and repositories of transformation tools. In this section we show how the semantics of the transformation modeling profile can be extended with the help of higher order transformations. This relates to the compilers or interpreters of transformation tools. In general, higher order transformations are defined as transformations that consume and/or produce transformation models. In the context of this paper, the role of higher order transformations is to transform transformation models that conform to an extension of the profile into transformation models that conform to the profile without the new language construct. The semantics of the core profile is assumed to be well-understood: it is merely a standard syntax for what has been applied for two decades already.

A key to the proposed integration architecture is that the higher order transformations themselves are modeled using the core profile for transformation modeling. Therefore, any transformation engine that supports the core profile can execute the higher order transformations. Consequently, any such engine can normalize transformation models that apply a new language construct into more primitive transformation models. In general, a transformation tool may execute a series of publicly available higher order transformations before executing the result on its native graph transformation engine. In the case of performance problems (that we have not encountered so far), some tools might directly generate optimized code for particular transformation language constructs. In that case, the higher order transformations may be used for checking the correctness of the optimized code.

In summary, higher order transformations ensure that new graph transformation language constructs can be executed by all tools that support the core

transformation modeling profile. On the one hand, this rewards the creator of the language construct with an instant adoption across a set of tools. On the other hand, this relieves tool builders from implementing constructs that have been defined by others.

4 Example Profile Extension: Copy Operator

This section explains how we contributed a *Copy* operator according to the proposed integration architecture. Subsection 4.1 describes the syntactical extension of the profile. Subsection 4.2 presents the higher order transformation that makes the operator executable in a tool-independent manner. Remark that copy operations can already be *programmed* with the core profile: one can specify match and create operations explicitly for all elements that need to be copied. For example, the rewrite rule from Figure 1 (b) explicitly enumerates which properties (*name* and *visibility*) from the *m2* node need to be copied to the *commonOperation* node. However, such transformation specifications are undesirably low-level. One would like to model declaratively which nodes should be copied without worrying about the attributes (cfr., *name*) and associations (cfr., *visibility*) of the related metaclasses (cfr., *Operation*).

4.1 Syntax

The need for a course-grained *Copy* operator has been acknowledged outside the graph transformation community as well [13]. Therefore, we propose first-class transformation language support for *modeling* the following activities:

- matching a *tree* pattern that defines the scope of the subgraphs that need to be copied,
- preserving internal edges that are not contained in that tree pattern,
- performing side-effects on the resulting copy: adding/removing nodes or edges and updating node attributes in the target subgraph.

Therefore, the following language constructs need to be added:

- the «*copy*» construct allows one to specify what node represents the entry point to the subgraph that needs to be copied.
- starting from such a «*copy*» node one can specify that an underlying tree pattern has *composition* semantics. Each node and edge matched by this pattern will be copied.
- the «*onCopy*» construct can be used to indicate that a particular instruction («*create*», «*destroy*», «*update*») needs to be executed on the copy of an element instead of on the element itself.
- the «*preserve-between-copies*» construct can be used to indicate that an edge in the subgraph needs to be copied to the target subgraph.

By packaging these three stereotypes («*copy*», «*onCopy*», «*preserve-between-copies*») in a library, any generic UML editor can be used to model copy operations concisely [14].

4.2 Semantics

This section describes the higher order transformation that normalizes the constructs presented in the previous section back into more primitive graph transformation constructs. Before presenting some fine-grained mapping rules between models from the *Copy* profile and those from the *core transformation* profile, we consider an example of an input and output transformation model. The input model contains the copy specific stereotypes presented in the previous section. The output model conforms to the core transformation profile. It is up to the higher order transformation to replace the declarative copy stereotypes by operational counter-parts from the core transformation profile.

Example Input Transformation Model. Figure 2 shows a rewrite rule from an example input transformation model. This rewrite rule applies the stereotypes presented in the previous section to model that a subgraph representing a *analysis model* needs to be copied into a subgraph representing a *design model*. The rule expresses that all classes, typed attributes, enumerations, inheritance links and association links need to be copied from one to the other subgraph. Classes within the design model are marked as persistent “entities”. We refer the reader to [14] for a more detailed description of the context and meaning of all elements. Also remark that this rewrite rule is part of a larger transformation model in which other rules take care of association class flattening [15], etc.

Example Output Transformation Model. Within the domain of the core transformation profile, there is no notion of the copy specific constructs. Therefore, the higher order transformation needs to turn the input transformation rules that apply these constructs into complex sequences of more primitive rewrite rules.

To make this more concrete, Figure 3 visualizes the sequence of rewrite rules by showing the control flow of the output transformation model. The 33 generated states are highlighted in clusters of light and dark grey. Each cluster represents rewrite rules for a specific type. States that were already contained

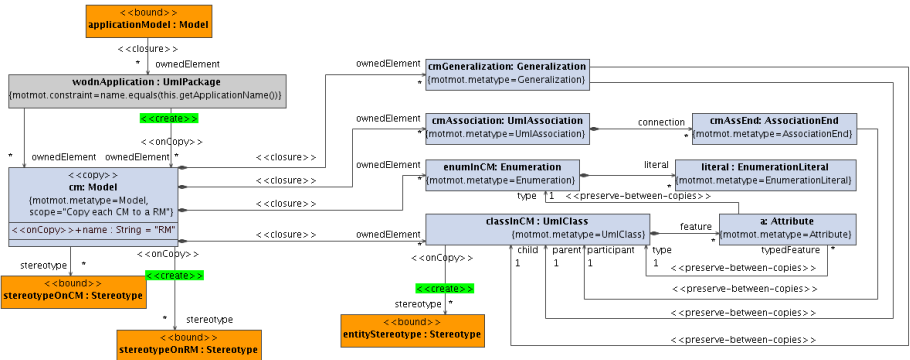


Fig. 2. Illustrative rewrite rule from an input first-order transformation model

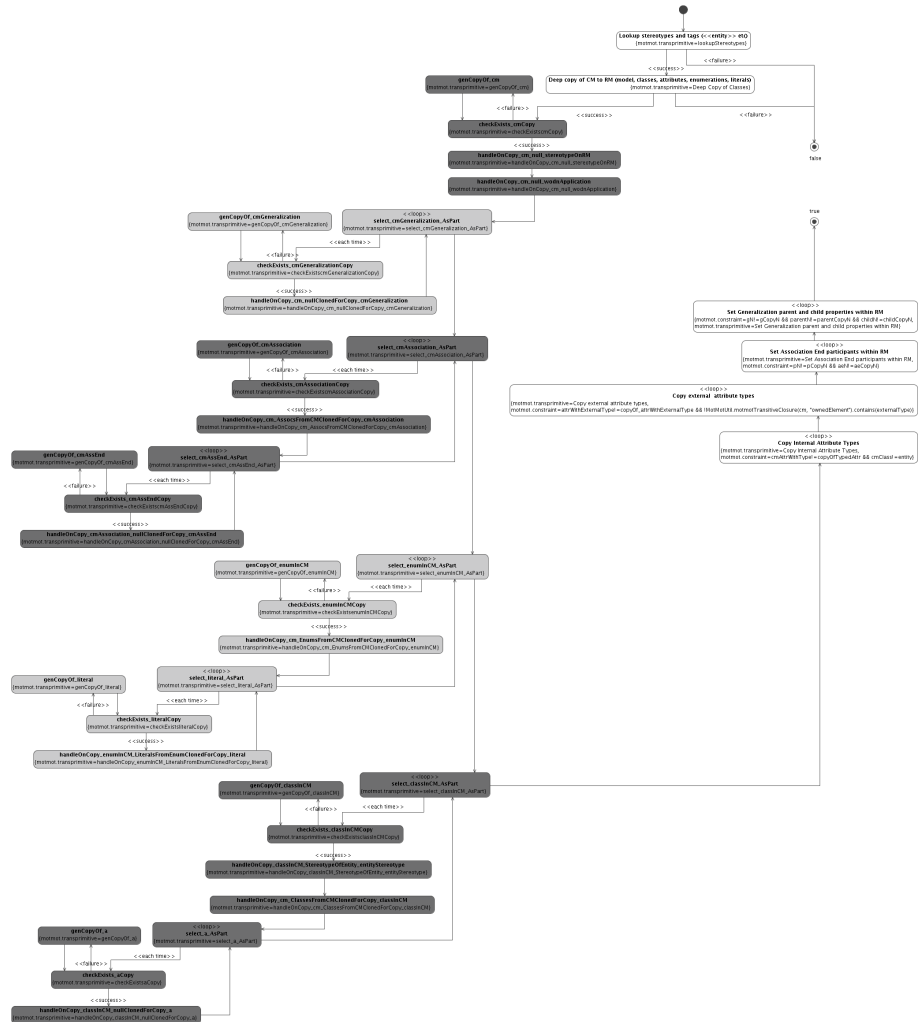


Fig. 3. Control flow of the output of the higher order transformation

in the input transformation model are shown in white. The layout of the diagram is designed as follows: the top-most cluster of dark-gray nodes contains all states required to copy the model element representing the analysis model. This element of type *Model* acts as a container for UML classes, enumerations, associations and generalizations. The four clusters of gray nodes that are displayed on the left of Figure 3 handle the copying of these contained elements:

- the upper cluster of light-gray nodes contains all states required to copy the contained elements of type *Generalization*,
- the following cluster of dark-gray nodes contains all states required to copy the contained elements of type *Association* and *AssociationEnd*,

- the following cluster of light-gray nodes contains all states required to copy the contained elements of type *Enumeration* and the contained *EnumerationLiteral* elements,
- the lower cluster of dark-gray nodes contains all states required to copy the contained elements of type *UmlClass* and the contained *Attribute* elements.

The order in which these clusters are executed could be altered without changing the behavior of the transformation. The diagram layout emphasizes the recursive nature of the transformation behavior. More specifically, each cluster follows a fixed pattern:

select an element from its container in the source subgraph,
check whether the element is already mapped to a copy,
generate a copy if needed,
manipulate the copied element,
recursively apply this pattern on the contained elements.

Figure 4 shows a fragment of Figure 3 in more detail. More specifically, the states related to the copying of enumerations are shown. The top right state (called *select_enumInCM_AsPart*) iterates over all *Enumeration* elements contained within the conceptual model. The second state checks whether such an element is already copied. The outgoing $\ll\text{failure}\gg$ transition ensures a copy is generated when needed.

When a copy is present, the outgoing $\ll\text{success}\gg$ transition ensures that the rewrite rule performing the side effects on the copy is triggered. As indicated by the state name, these manipulations of the copied elements relate to both the *cm* and the *enumInCM* nodes from the Story Pattern shown on Figure 2. More specifically, the state *handleOnCopy_cm_EnumsFromCMClonedForCopy_enumInCM* ensures that the copied enumerations are added to the

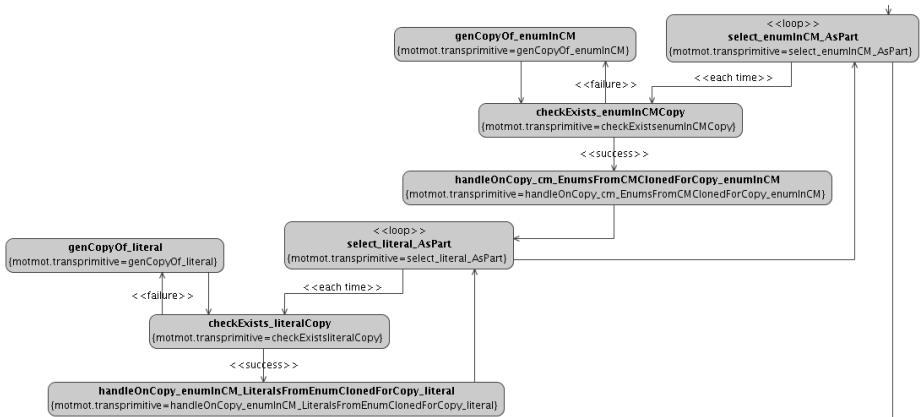


Fig. 4. Control flow of the rules for copying *Enumeration* and *EnumerationLiteral* elements

robustness model (which is a copy of the conceptual model). The automatically generated name of this state is based on the name of the edge between the *cm* and *enumInCM* nodes from the Story Pattern shown on Figure 2.

The transition to the *select_literal_AsPart* state realizes a recursive step. More specifically, the four states shown on the bottom left of Figure 4 realize the “*select, check, generate, manipulate*” pattern described above on all elements contained within an enumeration.

Figure 5 shows the rewrite rule for generating a copy of *Attribute* elements. The rule corresponds to a state in the lowest cluster from Figure 3. The higher order transformation generates such rewrite rules for all elements that need to be copied. It should be stressed that this generated rewrite rule conforms to the core transformation profile and can therefore be executed by any engine that implements that profile, even by those engines that have been implemented without knowledge of the *Copy* operator discussed in Section 4.1.

Since the nodes in the rewrite rules from the input transformation model (e.g., the *cm*, *classInCM*, *enumInCM*, *a*, ... nodes shown on Figure 2) do not contain these low-level attribute assignments explicitly, the higher order transformation needs to query some metamodel information. Using such metamodel information, the higher order transformation infers, for example, that the generated rewrite rule for *Attribute* nodes initializes the *copyOf_a_created* node with the following properties of the *a* node: *initialValue*, *name*, *visibility*, *isSpecification*, *multiplicity*, *changeability*, *targetScope*, *ordering* and *ownerScope*.

Figure 5 also illustrates that the output transformation model explicitly creates traceability links between the source and target subgraphs. Using that mechanism, the realization of the *Copy* operator supports change propagation from the source subgraph to its copy. Interestingly, such details are hidden by the transformation profile extension presented in Section 4.1.

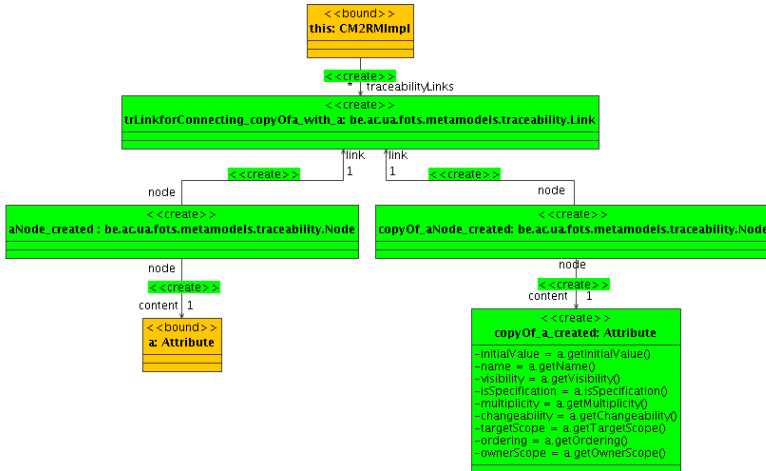


Fig. 5. Generated Story Pattern for Generating nodes of type *Attribute*

4.3 Example of a General Mapping Rule

This section generalizes the example application of the higher order transformation from the previous section into a mapping rule that is independent of the input first order transformation (such as the transformation from analysis to design models). The transformation model that realizes this mapping is publicly available in the MoTMoT project [12].

A rewrite node n_{copy} carrying the $\ll copy \gg$ construct expresses that its matched element needs to be copied. Within a rewrite rule, such a rewrite node should be preserved by the higher order transformation. Obviously, its $\ll copy \gg$ stereotype cannot be preserved in the output transformation model, but all other properties (state or node constraints, attribute assignments, etc.) are preserved. Additionally, a “*check and generate if needed*” pattern should be present in the output:

- a *check*-state, associated with a rewrite rule that models how the presence of an existing copy of the element can be checked,
- a $\ll failure \gg$ transition to a *generate*-state,
- a *generate*-state, whose rewrite rule models the actual creation of the copy,
- a transition back to the *check*-state.

Within the output transformation model, the rewrite rule modeling the *check*-state should contain a traceability link from a $\ll bound \gg$ node that represents the element from n_{copy} to another node of the same type. Similarly, the derived rewrite rule representing the *generate*-state should model the creation of a traceability link from the $\ll bound \gg$ representation of n_{copy} to the node representing the copy. Other mapping rules prescribe how the composition links, the $\ll onCopy \gg$, and the $\ll preserve-between-copies \gg$ constructs should be converted into constructs from the core transformation profile [10, Chapter 8].

5 General Application of the Approach

This section collects a representative set of challenges that need to be overcome for aligning the aforementioned tools (GReAT, MOFLON/TGG, VMTS, ...) with the proposed UML profile. On the one hand, these tool-specific issues should not be disregarded as trivial. On the other hand, this section illustrates that such issues are no fundamental obstacles to the adoption of the proposed approach either. The following sections focus on a well-known data-flow based language and a state-of-the-art mapping language, since such languages are sometimes perceived to be major variations on the graph transformation style that is captured by the profile presented in Section 2.

5.1 Data-Flow Constructs

The proposed transformation profile relies on activity diagrams as a control flow language. At first sight, this may seem incompatible with data-flow based

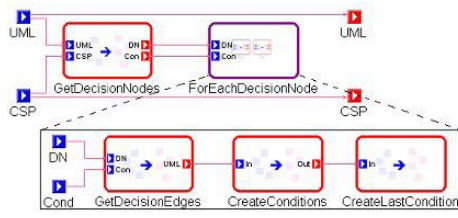


Fig. 6. GReAT data-flow for scheduling rewrite rules

scheduling structures in languages such as GReAT. Therefore, this section illustrates how a GReAT data-flow sequence can be mapped to a controlled graph transformation specification based on the profile’s activity diagram language.

Consider for example the data-flow specification shown on Figure 6. It controls the GReAT rules for mapping UML diagrams to CSP diagrams [16]. A distinctive feature of this specification style is the use of ports to transfer matched nodes across rewrite rules *explicitly*. In the proposed transformation profile, all nodes that have been matched by previously executed rules are made available automatically to subsequent rewrite rules. Thus, the two data transfer connectors between the *GetDecisionNodes* and *ForEachDecisionNode* blocks would not be modeled explicitly. Instead, one transition link would be modeled between two such activities. Additionally, the *DN* and *Con* nodes would be represented as *«bound»* nodes in rewrite rule corresponding to the the *ForEachDecisionNode* activity.

A second characteristic feature shown on Figure 6 is the support for iteration using hierarchical composition. More specifically, the *ForEachDecisionNode* block is an instance of a *ForBlock*. Therefore, each packet consumed by the *ForEachDecisionNode* block will be passed sequentially through the three lower level blocks. When mapping this configuration to the proposed profile, one would map the *ForEachDecisionNode* block to an activity marked with the *«loop»* construct. The three hierarchically composed blocks would be contained in an embedded *«each time»* flow.

A third issue that needs to be resolved is that GReAT graph elements can be renamed across a data-flow. For example, the output port of the *GetDecisionEdges* block shown on Figure 6 is called *UML*. By connecting it to the *In* port from the *CreateConditions* block, it is available under another name in the rewrite rule that corresponds to the latter block. Although this is not directly supported by the core version of the transformation modeling profile, it can be realized by means of a so-called *«alias»* construct [17].

A fourth, final but open issue is GReAT’s reliance upon the underlying C++ language for the specification of textual constraints and attribute updates. This issue is not specific to GReAT but is common for today’s graph transformation languages (e.g., the Fujaba tool relies on Java for similar purposes). Interestingly, an increasing number of languages is accompanied by a metamodel (e.g., a metamodel for Java has been released in the open source *NetBeans* project). Therefore, mappings from specialized expression languages (such as Imperative OCL, supported by VMTS [7]) into general purpose languages (such as C++ and

Java) can be supported by the proposed higher order transformation approach too. Before transformation tool builders generalize their expression language implementations, a limited amount of manual completion of transformation models will remain needed in practice.

In summary, although the GReAT language is based on data-flow modeling, it is close enough to the proposed profile to support the proposed integration architecture. The approach can be introduced incrementally: some language legacy constructs may initially require manual translation.

5.2 Bi-directional Rules

Triple Graph Grammars (TGGs) were introduced in the early nineties as a formalism for maintaining *bidirectional* consistency constraints between models originating from different software engineering tools [18]. The more recently proposed QVT *relations* strongly resemble triple graph grammar rules [9].

A triple rule not only consists of a left- and a right-hand side. Additionally, it divides the rewrite nodes and links in three domains: two domains represent the models that need to be kept consistent. A third domain represents the traceability model. Essentially, a triple rule describes the relations that need to hold between elements from the right-hand side when the elements from the left-hand side are already consistent.

Although TGG rules can be executed directly by a Java interpreter, their operational semantics is usually clarified by presenting the mapping of a TGG rule to conventional rewrite rules [19]. Burmester et al., for instance, map TGG rules to six primitive graph rewriting rules [20]: three rules for adapting changes to the source model and three for adapting changes to the the target model.

Such a mapping from triple rules into operational rules has already been realized by several transformation tools (e.g., MoRTen [21] and MOFLON [22]). Unfortunately, the syntax of the triple rules as well as that of the operational rules has always been formalized by a tool-specific metamodel. Moreover, the higher order transformation has always been implemented directly on the API of a Java or C(++) based tool. Therefore, the problems described in Section 1 have been exposed specifically in the domain of TGG tools too. As illustrated in [10], the proposed profile approach is applicable for standardizing the TGG syntax too. Moreover, the algorithm for deriving operational rules from triple rules can be realized using the proposed higher order transformation approach as well.

6 Related Work

The presented use of profiles and higher order transformations has not been proposed before for the integration of transformation languages. Nevertheless, the following references relate to particular aspects of the approach.

Standard Syntax for Rewrite Rules. First of all, Graph Transformation eXchange Language (GTXL [8]) has been proposed as a standard for exchanging transformation models. Unlike the proposed profile, GTXL has

no relation to a mainstream modeling language such as the UML. Therefore, there are no off-the-shelf industrial tools for editing GTXL models. Secondly, the GTXL metamodel relies on a XML DTD instead of on the MOF. Therefore, it requires more integration effort in an MDA tool integration context. Finally, GTXL only supports uncontrolled rules whereas the proposed profile supports rules that are controlled by activity diagrams. Finally, due to the lack of a profile concept, GTXL cannot be extended without breaking metamodel compatibility with its implementations. Secondly, the Queries/Views/Transformations (QVT [9]) standard presents three languages for transformation modeling. Apart from the MOF basis, it has the same limitations as GTXL. The QVT standard does promote bridges between its sublanguages by means of higher order transformations. In fact, the mapping between the relations and core language is formalized in the QVT relations language [9]. Unfortunately, the QVT relations language is not as generally applicable as the profile presented in this paper. Moreover, the complex semantics of the language requires more implementation effort and less semantical infrastructure can be reused through higher order transformations.

Higher Order Transformations. Within the VIATRA tool, the transformation process from human-oriented transformation models into machine-oriented transformation code is supported by higher order transformations too [23]. Unlike the proposed approach, the transformation models do not conform to any standards. Moreover, the higher order transformation is not written in a standard transformation language either.

When zooming out from the context of model transformation languages, one ends up in the realm of language integration in general. In that realm, the diversity of the language definition frameworks becomes much larger. More specifically, the supporting tools may no longer be defined by metamodels. Instead, they can be defined by attribute grammars. In that case, it is unclear how these tools should support the extension mechanism of a *profile*. Fortunately, in the research domain of attribute grammars one is mainly considering textual languages. Therefore, models can easily be exchanged in their concrete syntax directly. One is consequently not confronted with compatibility problems between repositories that give direct access to model elements or with problems between XMI files that are based on different types.

Still, integration issues arise that are similar to the ones presented in this paper. More specifically, one aims to extend the attribute grammar of a language, independently of a particular tool. Interestingly, one is also relying on transformations from an extended grammar to the grammar of the standard language. By using *forwarding* [24], one is able to package the rules for the language extensions in a separate module. These modules can be compared to our higher order transformations.

After years of experience in this area of language integration, Van Wyk indicates that it is often insufficient to normalize the constructs of language extensions in their standard counterparts. For example, the results of a verification

tool for the standard language may not be understandable to users of the extended language. Therefore, in some cases, he advises to write forwarding rules that are specific to a specific debugger or code generator. In [25], Van Wyk and Heimdahl describe how a debugger for a restricted language can be realized by means of a grammar that forwards to the core grammar for that language. The resulting debugger works for the restricted language as well as for extensions thereof. Then, they describe how the default debugging behavior can be improved by providing debugging rules for language extensions too. The authors describe a similar approach for improving PVS and NuSMV verification support for language extensions to the Lustre language.

Similar observations can be made in our approach: although debugging and verification tools for the core transformation language can be used directly on the output of our debugging-agnostic higher order transformations, the user experience is much better when debugging in terms of the language extensions. On the one hand, this is an area where tool builders can create added value. On the other hand, most of our higher order transformations create traceability links between high- and low-level constructs already. A tool can use this information to map low-level debugging elements back to their high level counter-parts. The tool does not become specific to one concrete higher order transformation or language extension. It only becomes specific to a metamodel for traceability. Fortunately, such a metamodel is very small (one or two metaclasses) and can be standardized when tool vendors want to commit to that.

7 Future Work

Once Fujaba, VMTS, MOLA, GReAT and Progres support the proposed metamodel discussed in Section 2.1, these tools will not only be able to exchange transformation models that apply the language constructs they supported already («*create*», «*destroy*», ...). Instead, they will also be able to load the stereotypes related to the declarative copying that was not anticipated when building these respective tools.

Remark once more that the *Copy* operator is only treated in so much detail to make the language extension approach as concrete as possible. A *Merge* or *Diff* operator (see [13]) could be defined similarly. In fact, we are actively working on profile extensions for negative application conditions and graph grammars with uncontrolled rule applications [26]. This work should make these popular AGG language constructs available to any transformation engine that supports the core profile [27].

In the proposed architecture, higher order transformations are realized using the same language as first-order transformations. This is enabled by using a transformation language that has a MOF metamodel. Although this is a simple technique in theory, the following practical issue still needs to be resolved: when using the MoTMoT prototype for the proposed architecture, we execute higher order transformations manually where needed. Ultimately, both the transformation profile extensions (stereotypes and tagged values) and the corresponding

higher order transformations are available in an online repository. Transformation tools should be able to access these artifacts automatically and apply the higher order transformations behind the scenes. Although the MoTMoT prototype provides an online build infrastructure for managing the *compilation*, *testing* and *deployment* and *versioning* of model transformations [28], a systematic process for distributing new versions of a higher order transformation to a public online repository has not been defined yet.

8 Conclusions

This paper presented a new approach to the integration of transformation languages and tools. A realization of the proposed approach enables transformation tool builders to focus on user-oriented added value (such as editor usability, run-time performance, etc.) and new, *declarative* language constructs, instead of spending time on the implementation of evaluation code that was already realized in other tools before. A unique characteristic of the approach is that it only requires transformation tool builders to implement a small core, and provides support for more declarative language constructs (bidirectional mapping, copying, ...) without breaking interoperability.

The approach relies on two techniques: first of all, its syntactic extensibility is based on the profile support of the metamodel of the “host” modeling language (e.g., the UML). Secondly, new language constructs are made executable by normalizing the profile extensions into the core profile by means of a higher order transformation that is modeled in the core profile itself. As a proof of concept, a core transformation profile was proposed as an OMG UML 1.5 profile.

The MoTMoT tool has already illustrated the executability and usefulness of that profile before. This prototype has now been used to contribute a *Copy* operator to the core profile, without writing any MoTMoT specific code. When somebody realizes *Merge*, *Diff*, or data-flow constructs using the same approach, any tool that supports the core profile will automatically be able to execute these constructs too.

References

1. Nagl, M., Schürr, A.: Summary and specification lessons learned. In: Nagl, M. (ed.) IPSEN 1996. LNCS, vol. 1170, pp. 370–377. Springer, Heidelberg (1996)
2. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the unified modeling language and java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000)
3. Jahnke, J.H., Schäfer, W., Wadsack, J.P., Zündorf, A.: Supporting iterations in exploratory database reengineering processes. *Sci. Comput. Program.* 45(2-3), 99–136 (2002)
4. University of Paderborn. Fujaba Tool Suite (2007), <http://www.fujaba.de/>
5. Agrawal, A.: Graph Rewriting And Transformation (GReAT): A solution for the Model Integrated Computing (MIC) bottleneck. In: ASE, p. 364 (2003)

6. Kalnins, A., Celms, E., Sostaks, A.: Simple and efficient implementation of pattern matching in MOLLA tool. In: 2006 7th International Baltic Conference on Databases and Information Systems, 3-6 July, pp. 159–167 (2006)
7. Levendovszky, T., Lengyel, L., Mezei, G., Charaf, H.: A systematic approach to metamodeling environments and model transformation systems in VMTS. *Electronic Notes in Theoretical Computer Science* 127, 65–75 (2005)
8. Lambers, L.: A new version of GTXL: An exchange format for graph transformation systems. *Electronic Notes in Theoretical Computer Science* 127, 51–63 (2005)
9. Object Management Group. MOF 2.0 QVT final adopted specifications (November 2005), <http://www.omg.org/cgi-bin/doc?ptc/05-11-01>
10. Van Gorp, P.: Model-driven Development of Model Transformations. PhD thesis, University of Antwerp (April 2008)
11. Green, T.R.G.: Cognitive dimensions of notations. In: Sutcliffe, A., Macaulay, L. (eds.) *People and Computers V*, New York, NY, USA, pp. 443–460. Cambridge University Press, Cambridge (1989)
12. Muliawan, O., Schippers, H., Van Gorp, P.: Model driven, Template based, Model Transformer (MoTMoT) (2007), <http://motmot.sourceforge.net/>
13. Bernstein, P.A.: Generic model management: A database infrastructure for schema manipulation. In: Batini, C., Giunchiglia, F., Giorgini, P., Mecella, M. (eds.) *CoopIS 2001*. LNCS, vol. 2172, pp. 1–6. Springer, Heidelberg (2001)
14. Van Gorp, P., Schippers, H., Janssens, D.: Copying Subgraphs within Model Repositories. In: Bruni, R., Varró, D. (eds.) *Fifth International Workshop on Graph Transformation and Visual Modeling Techniques*, Vienna, Austria. *Electronic Notes in Theoretical Computer Science*, pp. 127–139. Elsevier, Amsterdam (2006)
15. Gogolla, M., Richters, M.: Transformation rules for UML class diagrams. In: Bézivin, J., Muller, P.-A. (eds.) *UML 1998*. LNCS, vol. 1618, pp. 92–106. Springer, Heidelberg (1999)
16. Narayanan, A.: UML-to-CSP transformation using GReAT. In: *AGTiVE 2007 Tool Contest Solutions* (2007)
17. Van Gorp, P., Muliawan, O., Keller, A., Janssens, D.: Executing a platform independent model of the UML-to-CSP transformation on a commercial platform. In: Tüntzer, G., Rensink, A. (eds.) *AGTiVE 2007 Tool Contest* (January 2008)
18. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) *WG 1994*. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
19. Königs, A., Schürr, A.: Tool integration with Triple Graph Grammars - a survey. In: Heckel, R. (ed.) *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques*. *Electronic Notes in Theoretical Computer Science*, vol. 148, pp. 113–150. Elsevier, Amsterdam (2006)
20. Burmester, S., Giese, H., Niere, J., Tichy, M., Wadsack, J.P., Wagner, R., Wendehals, L., Zündorf, A.: Tool integration at the meta-model level: the Fujaba approach. *International Journal on STTT* 6(3), 203–218 (2004)
21. Wagner, R.: Consistency Management System for the Fujaba Tool Suite – MoTE/-MoRTen Plugins (January 2008), <https://dsd-serv.uni-paderborn.de/projects/cms/>
22. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In: Rensink, A., Warmer, J. (eds.) *ECMDA-FA 2006*. LNCS, vol. 4066, pp. 361–375. Springer, Heidelberg (2006)

23. Horváth, Á., Varró, D., Varró, G.: Automatic generation of platform-specific transformation. *Info-Communications-Technology LXI*(7), 40–45 (2006)
24. 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)
25. Van Wyk, E., Per Erik Heimdahl, M.: Flexibility in modeling languages and tools: A call to arms. *International Journal on STTT* (2009)
26. Meyers, B., Van Gorp, P.: Towards a hybrid transformation language: Implicit and explicit rule scheduling in Story Diagrams. In: *Proc. of the 6th International Fujaba Days 2008*, Dresden, Germany (September 2008)
27. Ermel, C., Rudolf, M., Taentzer, G.: The AGG approach: language and environment. In: *Handbook of graph grammars and computing by graph transformation. Applications, Languages, and Tools*, vol. II, pp. 551–603. World Scientific Publishing Co., Inc., River Edge (1999)
28. The Apache Software Foundation. Maven (2007), <http://maven.apache.org/>

Formalization and Rule-Based Transformation of EMF Ecore-Based Models

Bernhard Schätz

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
schaetz@in.tum.de

Abstract. With models becoming a common-place in software and systems development, the support of automatic transformations of those models is an important asset to increase the efficiency and improve the quality of the development process. However, the definition of transformations still is quite complex. Several approaches – from more imperative to more declarative styles – have been introduced to support the definition of such transformations. Here, we show how a *completely* declarative relational style based on the interpretation of a model as single structured term can be used to provide a transformation mechanism allowing a simple, precise, and modular specification of transformations for the EMF Ecore platform, using a Prolog rule-based mechanism.

1 Motivation

The construction of increasingly sophisticated software products has led to widening gap between the required and supplied productivity in software development. To overcome the complexity of realistic software systems and thus increase productivity, current approaches increasingly focus on a *model-based* development using appropriate description techniques (e.g., UML [1]). Here, the specification of a software product is described using *views* of the system under development (*horizontal*, e.g., structure, the behavior; *vertical*, e.g., component/sub-components). To combine those views, they are mapped onto a common model.

Especially in a model-based approach with structure-rich system descriptions automated development steps – using mechanized transformations – have the potential to provide an important technique to improve the efficiency of the development process. Besides increasing efficiency, these transformations can offer consistency ensuring modification of models, ranging from refactoring steps to improve the architecture of a system to the consistent integration of standard behavior. However, for their effective application, frameworks enabling these transformations should use formalisms to enable sufficiently abstract yet executable descriptions, support their modular definition by simple composition, and provide mechanisms for parameterization.

In the following, a transformation approach is introduced, allowing a simple, precise, and modular specification of transformations. The approach uses a declarative relational style to provide a transformation mechanism. It is implemented on the Eclipse platform, using a term-based abstract representation of an EMF Ecore description and a Prolog rule-based interpretation.

1.1 Related Approaches

The introduced transformation framework is used to describe graph transformations, using a relational calculus focused on basic constructs to manipulate nodes (elements) and edges (relations) of a conceptual model. In contrast to other graph-based approaches like MOFLON/TGG [2], VIATRA [3], FuJaBa [4], or GME [5] it is not based on triple graph-grammars or graphical, rule-based descriptions [6], but uses a textual description based on a relational, declarative calculus. As shown in Section 3, for the declaration of basic transformations the rule-based approach allows the definition of such transformations in a pre-model/post-model style of description similar to graph-patterns, both concerning complexity and readability of description. However, in contrast to those approaches, the approach introduced here uses only a single formalism to describe basic transformations as well as their compositions; thus, it avoids the problems of using different formalisms – like the passing of parameters between rules – to describe transformation patterns, like object diagrams, OCL expressions, and state machines. Similarly, this relational rule-based approach makes the order of application and interaction of basic transformations explicit, avoiding the imprecision resulting from the under-specification of these dependencies. This is especially important if the outcome of the transformation is essentially influenced by the order of rule application.

Considering the textual form of description, the transformation framework is similar to the QVT approach [7] and its respective implementations like ATL [8], F-Logics based transformation [9], or Tefkat [10]. However, in contrast to those it comes with a precise and straightforward definition of models *and* transformations. This definition is directly reflected in the description of models and transformations: There is only a single homogenous formalism with two simple construction/deconstruction operators to describe the basic transformation rules and their composition; complex analysis or transformation steps can be easily modularized since there are no side-effects or incremental changes during the transformation. Furthermore, the rule-based relational approach allows to arbitrarily mix more declarative and imperative forms of specification, which are strictly separated in those other approaches. Thus, e.g., it allows to construct an initial description of a specification in a purely declarative fashion, and to successively rephrase it into a more imperative and efficient form.

Similar to other approaches like VIATRA [3] or GEMS [11], Prolog is used to implement a relational style of describing transformations. However, in those approaches the model is encoded as a *set of base clauses*. As a consequence, transformations are implemented via rules based on side-effects using an *assert/retract*, thus making a truly relational description of transformations impossible. This use of side-effects, however, complicates the definition of complex declarative transformation, especially when describing pattern-like rules potentially requiring backtracking. A similar encoding is also used in formalizing models instantiating meta-models as facts in a relational data-based schemes, with the schemes derived from the meta-model relations, as in [12]. In contrast, here the model is encoded as a single complex *term*, which forms an input and output argument of the Prolog clause formalizing the transformation, thus allowing to use true relational declarative style without side-effects, even for complex transformation constructed in a modular fashion from simpler transformations. Note that the

approach is not restricted to the use of Prolog; also other formalisms like functional languages can be used.

Another advantage of the presented approach is its capability to interpret loose characterizations of the resulting model, supporting the exploration of a set of possible solutions. By making use of the back-tracking mechanism provided by Prolog, alternative transformation results can not only be applied to automatically search for an optimized solution, e.g., balanced component hierarchies, using guiding metrics; the set of possible solutions can also be incrementally generated to allow the user to interactively identify and select the appropriate solution.

Finally, the approach presented here addresses a different form of application: While most of the above approaches intend to support the transformation from one modeling domain to another (e.g., from class diagrams to DB schemata), here the goal is to provide also simple implementations for refactorings of models (e.g., combining a cluster of components). Nevertheless, the presented approach is of course also applicable to transformations between modeling domains.

1.2 Contribution

The approach presented in the following sections supports the transformation of EMF Ecore models using a completely *declarative relational* style in a *rule-based* fashion:

Relational: A transformation is described as a relation between the models before and after the transformation; models themselves are also described as relations.

Declarative: The description of the transformation relation characterizes the models before and after the transformation rather than providing an imperative sequence of transformation operations.

Rule-Based: The relations and their characterization of the models are described using a set of axioms and rules.

To support this style of description with its simple, precise, and modular specification of transformation relations on the problem- rather than the implementation-level, three essential contributions are supplied by the framework presented here:

1. The *term-structure representation* of models based on the structure of the meta-models they instantiate, as well its implementation in the Eclipse framework, providing the translation of models of EMF Ecore-based meta-models into their corresponding term-structure.
2. The *relational formalization of model transformations*, as well as its implementation using a Prolog rule-based interpretation, providing the execution of transformations within the Eclipse framework.
3. The *application of relations* as execution of – generally parameterized – transformations, as well as its implementation as an Eclipse plug-in, providing support for the definition and the execution of relational transformation rules.

These three contributions are described in the following sections. Section 2 provides a formalized notion of a model used in this relational approach as well as its representation in a declarative fashion using Prolog style. Section 3 describes the basic principles of describing transformations as relations, using rules similar to graph grammars as

a specific description style, and illustrates the modular composition of transformations. Section 4 illustrates the implemented tool support both for the definition of transformation and the transformation execution. Section 5 highlights some benefits and open issues.

2 Model Formalization and Structure

As mentioned in Section 1, the purpose of the approach presented here is the transformation of descriptions of systems under development to increase the efficiency and quality of the development process. According to the nomenclature of the OMG, the description of a system is called a (*system*) *model* in the following. Figure 1 shows such a model, as it is used in the AutoFOCUS [13] approach to describe the architectural structure of a system: the system *Context*, consisting of subcomponents *Sib*, *Comp*, and *Contr*, the latter with subcomponent *Sub*; the components have input and output ports like *InDst* and *InMid*, connected by channels like *InLeft*.

To construct formalized descriptions of a system under development, a ‘syntactic vocabulary’ – also called *conceptual model* in [13] – is needed. This conceptual model¹ characterizes all possible system models built from the *modeling concepts and their relations* used to construct a description of a system; typically, class diagrams are used to describe them. Figure 2 shows the conceptual elements and their relations used to describe the architectural structure of a system in the AutoFOCUS approach. These concepts are reflected in the techniques used to model a system. In the following subsection, a *formalization of conceptual models and system models* based on relations is given as well as their representation in a declarative fashion using Prolog style.

2.1 Formalization

To define the notion of a *conceptual model* in the context of model transformations more formally, the interpretation along the lines of [13] is used. It provides a semantical interpretation for syntactical descriptions like in Figure 2. Basically, the conceptual model is constructed from a *conceptual universe*, containing the primitives used to describe a system: *concepts* characterizing unique entities used to describe a system, with examples in the AutoFOCUS conceptual model like *component*, *port*, or *channel* to define the components, ports and channels of a structural description of a system; *attributes* characterizing properties, like *name* or *direction* to define the name of

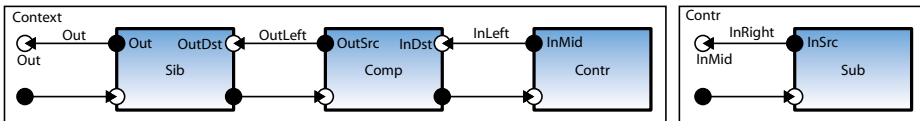


Fig. 1. Example: Hierarchical Component Model

¹ In the context of technologies like the Meta Object Facility, the class diagram-like definition of a conceptual model is generally called *meta model*.

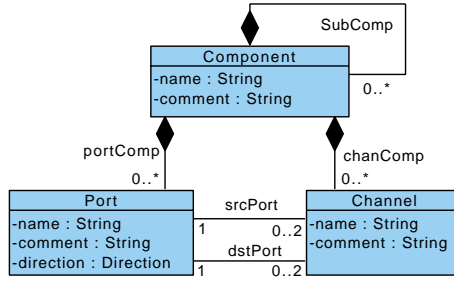


Fig. 2. Representation of the Conceptual Model of AutoFOCUS Component Diagrams

a component and a channel, or the direction (input or output) of a port. Concepts and attributes form the conceptual universe, consisting of a collection of infinite sets of conceptual entities, and a collection of – finite or infinite – sets of attribute values. In case of the AutoFOCUS conceptual model, examples for sets of conceptual entities are $CompId = \{comp_1, comp_2, \dots\}$, and $PortId = \{port_1, port_2, \dots\}$; typical examples for set of attribute values are $CompName = \{\text{'Sib'}, \text{'Context'}, \dots\}$ or $PortDir = \{\text{'input'}, \text{'output'}\}$.

Based on the conceptual universe, the *conceptual domain* is defined, consisting of elements corresponding to objects used to model a system, like *Component*, *Port*, or *Channel* to define the components, ports and channels of a structural description of a system; and relations corresponding to dependencies between the elements, like *subComp*, *chanComp*, or *srcPort* to define the subcomponents of a component, the component containing a channel, or the source port of channel.

The conceptual domain consists of a collection of element relations between conceptual entities and attribute values, and a collection of (binary) association relations between conceptual entities. In case of the AutoFOCUS conceptual domain for structural descriptions as provided in Figure 2², examples for element relations are $Component = CompId \times CompName$ with values $\{(comp_1, \text{'Context'}), (comp_2, \text{'Sib'}), \dots\}$, $Port = PortId \times PortName \times PortDir$ with values $\{(port_1, \text{'OutSrc'}, \text{'output'}), \dots\}$, or $Channel = ChanId \times ChanName$ with values $\{(channel_1, \text{'OutLeft'}), \dots\}$; examples for association relations are $SubComp = CompId \times CompId$ with values $\{(comp_1, comp_2), \dots\}$ or $srcPort = ChanId \times PortId$ with values $\{(channel_1, port_1), \dots\}$. Intuitively, the conceptual domain describes the domain, from which specific instances of the description of an actual system are constructed.

Based on the conceptual domain, the conceptual model is the set of all possible system models that can be constructed within this domain. Intuitively, each system model is a “sub-model” of the conceptual domain, with sub-sets of its entities and relations. In order to be a proper system model, a subset of the conceptual domain generally must fulfil additional constraints; typical examples are the constraints in meta-models represented as class diagrams. In case of the AutoFOCUS conceptual model shown in Figure 2, e.g., each port or channel element must have an associated component in

² For simplification purposes, the *Comment* attribute is ignored in the following.

the `chanComp` and `portComp` relation, resp.; furthermore, each channel must have an associated source and destination port in the `srcPort` and `dstPort` relation.

2.2 Structure of the Model

The transformation framework provides mechanisms for a pure (i.e., side-effect free) declarative, rule-based approach to model transformation. To that end, the framework provides access to EMF Ecore-based models [14]. As described in Subsection 2.1, formally, a (conceptual) model is a collection of sets of elements (each described as a conceptual entity and its attribute values) and relations (each described as a pair of conceptual entities). To syntactically represent such a model, a Prolog term is used. Since these elements and relations are instances of classes and associations taken from an EMF Ecore model, the structure of the Prolog term – representing an instance of that model – is inferred from the structure of that model. The term comprises the classes and associations, of which the instance of the EMF Ecore model is constructed. It is grouped according to the structure of that model, depending on the package structure of the model and the classes and references of each package. The structure of the model is built using only simple elementary Prolog constructs, namely compound functor terms and list terms. Note that this construction obviously is not specific to Prolog; the same approach can be used for the embedding in other rule-based formalisms, e.g., functional languages like Haskell [15].

To access a model, the framework provides construction predicates to deconstruct and reconstruct a term representing a model. Since the structure makes only use of compound functor terms and list terms, only two classes of construction predicates are provided, namely the union operation and the composition operations.

Term Structure of the Model. A *model term* describes the content of an instance of a EMF Ecore model. Each model term is a list of package terms, one for each packages of the EMF Ecore model. Each *package term*, in turn, describes the content of the package instance. It consists of a functor – identifying the package – with a sub-packages term, a classes terms, and an associations term as its argument. The sub-packages term describes the sub-packages of the package; it is a list of package terms.

The *classes term* describes the EClasses of the corresponding package. It is a list of class terms, one for each EClass of the package. Each *class term* consists of a functor – identifying the class - and an elements term. An *elements term* describes the collection of objects instantiating this class, and thus – in turn – is a list of element terms. Note that each elements term comprises only the collection of those objects of this class, which are not instantiations of subclasses of this class; objects instantiating specializations of this class are only contained in the elements terms corresponding to the most specific class. Finally, an *element term* - describing such an instance – consists of a functor – again identifying the class this object belongs to – with an entity identifying the element and attributes as arguments. Each of the attributes are atomic representations of the corresponding values of the attributes of the represented object. The entity is a regular atom, unique for each element term.

Similarly to an elements term, each *associations term* describes the associations, i.e., the instances of the EReferences of the EClasses, for the corresponding package. Again,

Table 1. The Prolog Structure of a Model Term

<i>ModelTerm</i>	::=	[<i>PackageTerm</i> (, <i>PackageTerm</i>) [*]]
<i>PackageTerm</i>	::=	<i>Functor</i> (<i>PackagesTerm</i> , <i>ClassesTerm</i> , <i>AssociationsTerm</i>)
<i>PackagesTerm</i>	::=	[] [<i>PackageTerm</i> (, <i>PackageTerm</i>) [*]]
<i>ClassesTerm</i>	::=	[] [<i>ClassTerm</i> (, <i>ClassTerm</i>) [*]]
<i>ClassTerm</i>	::=	<i>Functor</i> (<i>ElementsTerm</i>)
<i>ElementsTerm</i>	::=	[] [<i>ElementTerm</i> (, <i>ElementTerm</i>) [*]]
<i>ElementTerm</i>	::=	<i>Functor</i> (<i>Entity</i> (, <i>AttributeValue</i>) [*])
<i>Entity</i>	::=	<i>Atom</i>
<i>AttributeValue</i>	::=	<i>Atom</i>
<i>AssociationsTerm</i>	::=	[] [<i>AssociationTerm</i> (, <i>AssociationTerm</i>) [*]]
<i>AssociationTerm</i>	::=	<i>Functor</i> (<i>RelationsTerm</i>)
<i>RelationsTerm</i>	::=	[] [<i>RelationTerm</i> (, <i>RelationTerm</i>) [*]]
<i>RelationTerm</i>	::=	<i>Functor</i> (<i>Entity</i> , <i>Entity</i>)

it is a list of association terms, with each *association term* consisting of a functor – identifying the association – and an relations term, describing the content of the association. The *relations term* is a list of relation terms, each *relation term* consisting of a functor – identifying the relation – and the entity identifiers of the related objects. In detail, the Prolog model term has the structure shown in Table 1 in the BNF notation with corresponding *non-terminals* and *terminals*.

The functors of the compound terms are deduced from the EMF Ecore model, which the model term is representing:

- the functor of a *PackageTerm* corresponds to the name of the EPackage the term is an instance of;
- the functor of a *ClassTerm* to the name of the EClass the term is an instance of; and finally
- the functor of an *AssociationTerm* corresponds to the name of the EReference the term is an instance of.

Since EMF – unlike MOF – does not support associations as first-class concepts like EClasses but uses EReferences instead, EReference names are not necessarily unique within a package. Therefore, if present in the ECore model, EAnnotation attributes of EReferences are used as the functors of an *AssociationTerm*. Similarly, the atoms of the attributes are deduced from the instance of the EMF Ecore model, which the model term is representing:

- the entity atom corresponds to the object identifier of an instance of a EClass, while
- the attribute corresponds to the attribute value of an instance of an EClass.

Currently, while basically also multi-valued attributes can be handled by the formalism, only single-valued attributes like references (including null references), basic types, and enumerations are supported by the implementation.

2.3 Construction Predicates

In a strictly declarative rule-based approach to model-transformation, the transformation is described in terms of a predicate, relating the models before and after the transformation. Therefore, mechanisms are needed in form of predicates to deconstruct a model into its parts as well as to construct a model from its parts. As the structure of the model is defined using only compound functor terms and list terms, only two forms of predicates are needed: union and composition operations.

List Construction. The construction and deconstruction of lists is managed by means of the union predicate `union/3` with template³

```
union(?Left, ?Right, ?All)
```

such that `union(Left, Right, All)` is true if all elements of list `All` are either elements of `Left` or `Right`, and vice versa. Thus, e.g., `union([1, 3, 5], R, [1, 2, 3, 4, 5])` succeeds with `R = [2, 4]`.

Compound Construction. Since the compound structures used to build the model instances depend on the actual structure of the EMF Ecore model, only the general schemata used are described. Depending on whether a package, class/element, or association/relation is described, different schemata are used. In all three schemata the name of the package, class, or relation is used as the name of the predicate for the compound construction.

Package Compounds. The construction and deconstruction of packages is managed by means of package predicates of the form `package/4` with template

```
package(?Package, ?Subpackages, ?Classes, ?Associations)
```

where `package` is the name of the package (de)constructed. Thus, e.g., a package named `structure` in the EMF Ecore model is represented by the compound constructor `structure`. The predicate is true if `Package` consists of subpackages `Subpackages`, classes `Classes`, and associations `Associations`. The predicate is generally used in the form `package(+Package, -Subpackages, -Classes, -Associations)` to deconstruct a compound structure into its constituents, and `package(-Package, +Subpackages, +Classes, +Associations)` to construct a compound structure from its constituents.

Class and Element Compounds The (de)construction of classes/elements is managed by means of class/element predicates of the form `class/2` and `class/N+2` where `N` is the number of the attributes of the corresponding class, with templates

```
class(?Class, ?Elements)
class(?Element, ?Entity, ?Attribute1, ..., ?AttributeN)
```

³ According to standard convention, arbitrary/input/output arguments of predicates are indicated by `?/+/-`.

where class is the name of the class and element (de)constructed. Thus, e.g., the class named `Component` in the EMF Ecore model in Figure 2 is represented by the compound constructor `Component`. The class predicate is true if `Class` is the list of `Objects`; it is generally used in the form `class(+Class, Objects)` to deconstruct a class into its list of objects, and `class(-Class, +Objects)` to construct a class from a list of objects. Similarly, the element predicate is true if `Element` is an `Entity` with attributes `Attribute1, ..., AttributeN`; it can be used to deconstruct an element into its entity and attributes via `class(+Element, -Entity, -Attribute1, ..., -AttributeN)`, to construct an element from an entity and attributes (e.g. to change the attributes of an element) via `class(-Element, +Entity, +Attribute1, ..., AttributeN)`, or to construct an element including its entity from the attributes via `class(-Element, -Entity, +Attribute1, ..., AttributeN)`. Thus, e.g., `Component(Components, [Context, Sib, Contr])` is used to construct a class `Components` from a list of objects `Context`, `Sib`, and `Contr`. Similarly, `Component(Contr, Container, "Contr", "The container element")` is used to construct an element `Contr` with entity `Container`, name `"Contr"`, and comment `"The container element"`.

Association and Relation Compounds The construction and deconstruction of associations and relations is managed by means of association and relation predicate of the form `association/2` and `association/3` with templates

```
association(?Association, ?Relations)
association(?Relation, ?Entity1, ?Entity2)
```

where `association` is the name of the association and relation constructed/deconstructed. Thus, e.g., a relation named `subComponent` in the EMF Ecore model in Figure 2 is represented by the compound constructor `subComponent`. The relation predicate is true if `Association` is the list of `Relations`; it is generally used in the form `association(+Association, -Relations)` to deconstruct an association into its list of relations, and `association(-Association, +Relations)` to construct an association from a list of relations. Similarly, the relation predicate is true if `Relation` associates `Entity1` and `Entity2`; it is used to deconstruct a relation into its associated entities via `association(+Relation, -Entity1, -Entity2)` and to construct a relation between two entities via `association(-Relation, +Entity1, +Entity2)`. E.g., `subComponent(subComps, [CtxtSib, CtxtComp, CtxtContr])` is used to construct the subcomponent association `subComps` from the list of relations `CtxtSib`, `CtxtComp`, and `CtxtContr`. Similarly, `subComponent(CtxtContr, Context, Contr)` is used to construct relation `CtxtContr` with `Contr` being the subcomponent of `Context`.

3 Transformation Definition

The conceptual model and its structure defined in Section 2 was introduced to define transformations of system models as shown in Figure 1, in order to improve

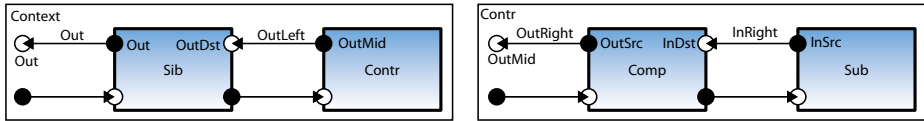


Fig. 3. Example: Result of Pushing Down Component *Comp* into Container *Contr*

the development process by mechanized design steps. A typical transformation step is the pushing down of a component into a container component, making it a subcomponent of that container. Figure 3 shows the result of such a transformation pushing down component *Comp* of the system in Figure 1 into the container *Contr*. Besides changing the super component of *Comp* from *Context* to *Contr*, furthermore channel chains (e.g., *InLeft*, *InRight*)⁴ from a port of a subcomponent (e.g., *Sub*) of the container (e.g., *InSrc*) via an intermediate port of the container (e.g., *InMid*) to a port of *Comp* (e.g., *InDst*) have to be shortened by eliminating the first channel and the intermediate port; symmetrically, a channel (e.g., *OutLeft*) from a port of the component (e.g., *OutSrc*) to a port (e.g., *OutDst*) of a sibling (e.g., *Sib*) of the component have to be extended by an additional channel (e.g., *OutRight*) and an intermediate port (e.g., *OutMid*).

In a relational approach to the description of model transformations, such a transformation is described as a relation between the model prior to the transformation (e.g., as given in Figure 1) and the model after the transformation (e.g., as given in Figure 3). In this section, the basic principles of describing transformations as relations are described in Subsection 3.1, while Subsection 3.2 shows how rules similar to graph grammars can be used as a specific description technique, by describing a transformation by partitioning a pre-model into parts eliminated and parts retained in the post-model, and partitioning the post-model into parts retained from the pre-model and parts added. Of course, also other, more operational styles are also supported by the rule-based approach, using, e.g., condition-branches and loop-like constructs.

3.1 Transformations as Relations

In case of the push-down operation, the relation describing the transformation has the interface

```
pushpull (PreModel, Comp, Contr, PostModel)
```

with parameter *PreModel* for the model before the transformation, parameter *PostModel* for the model after the transformation, and parameters *Comp* and *Contr* for the component of the model to be pushed down and the component to contain the pushed-down component, respectively. In the relational approach presented here, a transformation is basically described by breaking down the pre-model into its constituents and build up the post-model from those constituents using the relations from Section 2, potentially adding or removing elements and relations.

⁴ As shown in Figures 4 and 5, *Left* and *Right* refers to the elements of a channel-chain *left* (or “up-stream”) and *right* (or “down-stream”) of a split or merge point.

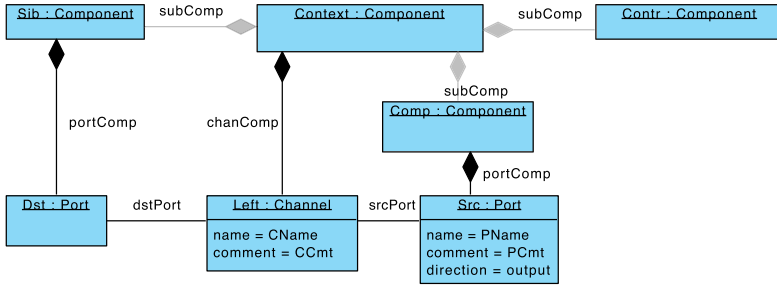


Fig. 4. Precondition of the Split-Merge-Transformation

With `PreModel` taken from the conceptual domain described in Figure 2 and packaged in a single package *structure* with no sub-packages, it can be decomposed in contained classes (e.g., *Port* and *Channel*) and associations (e.g., *portComp* and *srcPort*) via

```
structure(PreModel,PrePackages,PreClass,PreAssoc),
union([PreComps,PrePorts,PreChans],[],PreClass),
  channel(PreChans,InChans),port(PrePorts,InPorts),
union([PrePortComp,PreSrcPort],[],PreAssoc),
  portComp(PrePortComp,InPortComp),srcPort(PreSrcPort,InSrcPort)
```

In the same fashion, `PostModel` can be composed. Besides using the basic relations to construct and deconstruct models (and add or remove elements and relations, as shown in the next subsection), new relations can be defined to support a modular description of transformation, decomposing rules into sub-rules. E.g., in the `pushpull` relation, the transformation can be decomposed into the reallocation of the component and the rearrangement of the channel-chains; for the latter, then a sub-rule `splitmerge` is introduced, as explained in the next subsection.

3.2 Pattern-Like Rules

To define the transformation steps for splitting and merging channels during the push-down operation, we use a relation `splitmerge`, merging/splitting a channel starting or ending at a port of the pushed-down component `Comp`. This relation – formalized as a Prolog predicate `splitmerge` – with interface

```
splitmerge(
  InPorts, InChans, InPortComp, InChanComp, InSrcPort, InDstPort,
  Context, Comp, Contr,
  OutPorts, OutChans, OutPortComp, OutChanComp, OutSrcPort, OutDstPort)
```

relates the *Port* and *Channel* elements as well as the *PortComp*, *ChanComp*, *SrcPort*, and *DstPort* associations of the models before and after the transformation, depending on the component `Comp` to be relocated, its container component `Context`, and the component `Contr` it is pushed into. The splitting/merging of a channel connected to `Comp` depends on whether the channel is an input or an output channel of `Comp`, and

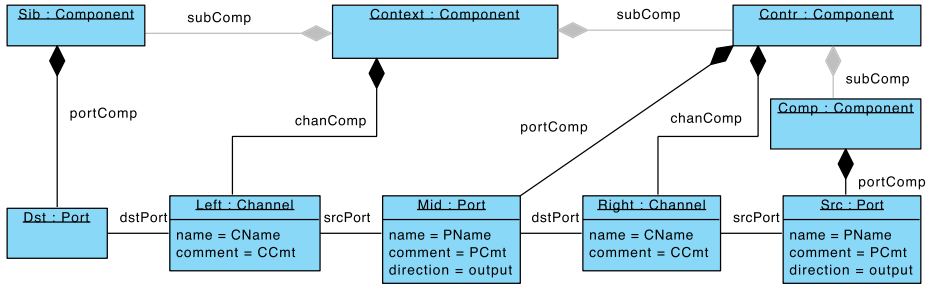


Fig. 5. Postcondition of the Split-Merge-Transformation

whether it is connected to `Contr` or a sibling component within `Context`. Therefore, in a declarative approach, we introduce four different – recursive – split/merge rules for those four cases, each with the interface described above. Furthermore, a fifth rule is added, covering the case that no channels have to be split/merged, thus ensuring the termination of the recursive rule application.

To define the bodies of those rules, we use a pattern-like formalization, describing conditions over the models before and after the transformation step. These conditions are described in form of element and association terms that must be present in these models. Figures 4 and 5 give a graphical representation of these conditions in form of object diagrams – defining the transformation rule for splitting an output channel of `Comp` connected to a sibling component `Sib` – by describing the elements and relations that must be contained in the model before and after the transformation.⁵ Elements and relations present in the precondition in Figure 4 but not in the postcondition in Figure 5 are not kept during the transformation and therefore effectively removed, elements and relations present in the postcondition but not in the precondition are added during transformation, all others are left unchanged during the transformation.

Figure 4 states that before the transformation, `Comp` and `Sib` each have an associated port `Src` and `Dst`, connected by channel `Left`, owned by `Context`. Figure 5 states that after the transformation, `Comp` is a subcomponent of `Contr`; furthermore, a port `Mid` of component `Contr` must be present with the same attributes as `Src`, as well as a channel `Right` owned by `Contr`, with `Left` connecting `Mid` and `Dst`, and `Right` connecting `Src` and `Mid`.

To define this transformation, the conceptual model and its structured representation introduced in Section 2 are used. Figure 6 shows the relational rule-based formalization of this step by giving the Prolog body of the corresponding clause; underlined relations correspond to the construction predicates from Section 2.⁶ Note that this single rule combines the information of *both graphical representations* in Figures 4 and 5; it furthermore also describes the order in which individual split-merge steps are chained

⁵ Grayed-out associations are not used in the patterns but added only for clarification.

⁶ For ease of reading, functors for element relations from Figure 2 with capitalized identifiers like `Port` or `Channel` are written as `port` or `channel`; in the actual application, the versions '`Port`' and '`Channel`' are used.

```

1  union([SrcComp,DstSib],InPortComp,PrePortComp),
2  portComp(SrcComp,Src,Comp),portComp(DstSib,Dst,Sib),
3  union([LeftSrc],InSrcPort,PreSrcPort),srcPort(LeftSrc,Left,Src),
4  union([LeftDst],InDstPort,PreDstPort),dstPort(LeftDst,Left,Dst),
5  union([SrcPort],InPorts,PrePorts),port(SrcPort,Src,PName,PComment,output),
6  union([LeftChan],InChans,PreChans),channel(LeftChan,Left,CName,CComment),
7  union([LeftCxt],InChanComp,PreChanComp),chanComp(LeftCxt,Left,Context),
8  splitmerge(InPorts,InChans,InPortComp,InChanComp,InSrcPort,InDstPort,
9  Context,Comp,Contr,
10 OutPorts,OutChans,OutPortComp,OutChanComp,OutSrcPort,OutDstPort),
11 chanComp(RightCnt,Right,Contr),union([LeftCxt,RightCnt],OutChanComp,PostChanComp),
12 channel(RightChan,Right,CName,CComment),union([LeftChan,RightChan],OutChans,PostChans),
13 port(MidPort,Mid,PName,PComment,output),union([SrcPort,MidPort,DstPort],OutPorts,PostPorts),
14 dstPort(LeftMid,Left,Mid),dstPort(RightDst,Right,Dst),
15 union([LeftMid,RightDst],OutDstPort,PostDstPort).
16 srcPort(LeftSrc,Left,Src),srcPort(RightMid,Right,Mid),
17 union([LeftSrc,RightMid],OutSrcPort,PostSrcPort),
18 portComp(MidCnt,Mid,Contr),union([SrcComp,MidCnt,DstSib],OutPortComp,PostPortComp).

```

Fig. 6. A Split-Merge Rule for Sibling Channels

together in a modular fashion, which is especially important in transformations where the order of application influences the outcome. Within a graphical specification of transformations, this later step cannot be described using those object-diagram-like diagrams at all. Therefore, graphical specifications require to use additional forms of diagrams, e.g., state-transition diagrams as used in [3], [4], or [5]. Unlike those, the approach here allows to simply pass information in form of parameters from the execution of one rule to the next.

Lines 1 to 7 describe the conditions before the transformation, and thus formalize the conditions described in Figure 4. Similarly, lines 11 to 18 describe the conditions after the transformation, thus formalizing the conditions described in Figure 5. Since the relational rule-based approach allows the use of recursively defined transformations, lines 8 to 10 introduce further transformation steps after removing the elements of the precondition and before introducing the elements of the postcondition. Lines 1 to 7 use a common scheme to ensure the validity of the conditions for the elements and relations taking part in the transformation: for each element and relation

- its occurrence in the list of corresponding elements/relations of the pre-model is established via the union constructor
- its contained entities and attributes within the pre-model are established via its corresponding functor

Thus, e.g., line 5 ensures the existence of a port element `SrcPort` consisting of port entity `Src` and attributes `PName`, `PComment`, and `output`, while line 7 ensures the existence of a `chanComp` relation `LeftCxt` linking channel `Left` to component `Context`. The union constructor is not only used to establish the occurrence of elements and relations in the pre-model; it is furthermore used to remove the identified elements and relations from the pre-model. E.g., line 5 also removes the port element `SrcPort` from the port class `PrePorts` of the pre-model before assigning the remainder to the port class `InPorts`, which is used in line 8 as input parameter of the recursive application of the `splitmerge` relation.

```

1  union([SrcComp,MidContr,DstSib],InPortComp,PrePortComp),portComp(SrcComp,Src,Comp),
2  portComp(MidContr,Mid,Contr),portComp(DstSib,Dst,Sib),
3  union([RightSrc,LeftMid],InSrcPort,PreSrcPort),
4  srcPort(RightSrc,Right,Src),srcPort(LeftMid,Left,Mid),
5  union([RightMid,LeftDst],InDstPort,PreDstPort),
6  dstPort(RightMid,Right,Mid),dstPort(LeftDst,Left,Dst),
7  union([SrcPort,MidPort,DstPort],InPorts,PrePorts),port(SrcPort,Src,_,_,output),
8  port(MidPort,Mid,_,_,output),
9  union([LeftChan,RightChan],InChans,PreChans),
10 channel(LeftChan,Left,_,_),channel(RightChan,Right,_,_),
11 union([LeftCxt,RightCnt],InChanComp,PreChanComp),chanComp(LeftCxt,Left,Context),
12 chanComp(RightCnt,Right,Contr),
13 splitmerge(InPorts,InChans,InPortComp,InChanComp,InSrcPort,InDstPort,
14 Context,Comp,Contr,
15 OutPorts,OutChans,OutPortComp,OutChanComp,OutSrcPort,OutDstPort),
16 union([LeftCxt],OutChanComp,PostChanComp),
17 union([LeftChan],OutChans,PostChans),
18 union([SrcPort],OutPorts,PostPorts),
19 union([LeftDst],OutDstPort,PostDstPort),
20 srcPort(LeftSrc,Left,Src),union([LeftSrc],OutSrcPort,PostSrcPort),
21 union([SrcComp,DstSib],OutPortComp,PostPortComp).

```

Fig. 7. A Split-Merge Rule for Container Channels

The scheme of lines 1 to 7 is used in symmetrical fashion in lines 11 to 18 to ensure the validity of the conditions for the elements and relations of the post-model. Thus, e.g., line 13 ensures that a port element `MidPort`, consisting of port entity `Mid` and attribute values defined by the variables `PName` and `PComment`, as well as the constant output of enumeration `Direction`, exists within port class `PostPorts`; similarly, line 11 ensures that a `chanComp` relation `RightCnt`, linking channel `Right` to component `Contr`, exists within `compChan` association `PostChanComp`, together with relation `LeftCxt`.

In a purely relational interpretation of declarative transformations, the ordering of the relations used to define a rule is irrelevant. Thus, using that kind of interpretation, a transformation can be applied in two ways. E.g., in case of the `pushpull`-transformation, the relation can be used to push-down a component into its container by assigning the pre- and post-models to the `PreModel` and `PostModel` parameter, respectively. Symmetrically it can be used to pull-up a component from a container by assigning the pre-model to the `PostModel` and the post-model to the `PreModel` parameter. As however the Prolog engine interpreting the declarative transformation rules uses a sequential evaluation of the relations within a rule, the ordering becomes decisive concerning the *efficiency of the application* of the transformation. Thus, in the rules of Figures 6 and 7, the relations are ordered in correspondence with the binding of variables. E.g., in Figure 6 the binding extends from `Comp` over `Src` and `Left` to `PName`, `PComment`, `CName`, and `CComment`.

Figure 7 shows a variant form of the `splitmerge` relation.⁷ While the previous variant deals with the introduction of an additional channel connecting `Comp` to a sibling component `Sib` when pushing it down into `Contr`, this variant deals with the

⁷ For ease of reading, the anonymous variable “_” is used in the position of irrelevant attributes.

elimination of a channel connecting `Comp` to a subcomponent `Sub` of `Contr`. Again, lines 1 to 12 define the preconditions of the transformation, lines 16 to 21 define the postconditions, and lines 13 to 15 make use of the recursively defined `splitmerge` relation.

Since the two `splitmerge` rules are inverse transformation of each other, they can be essentially obtained by exchanging element and relation terms of the pre-models with those of the post-model, and the reordering of the clauses. Thus, e.g., the union-term of line 1 of Figure 6 corresponds to the union-term of line 21 of Figure 7 as well as the union-term of line 18 of the former corresponds to the union-term of line 1 of the latter, when swapping `InPortComp` and `PrePortComp` with `OutPortComp` and `PostPortComp`, resp. Similarly, the `chanComp`-functor terms in Figure 6 of lines 7 and 11 correspond to the `chanComp`-functor terms in Figure 7 of lines 16 and 12.

Basically, by swapping the characterization of pre- and post-model, inverting the sequence of predicates, an efficient version of `splitmerge` for the introduction of a channel is turned into an efficient version for the elimination of a channel. The reordering is based on the general principle to let predicates with a more bound arguments precede predicates with less bound arguments. While some aspects of ordering are essential to obtain a feasible execution – especially putting the characterization of the pre-model before the recursive call and the characterization of the post-model after it – other reorderings – especially of predicates with no shared variables – have no or little effect on the usability of the execution.

4 Tool Support

The presented approach has been implemented as an Eclipse plugin using the `tuProlog` engine [16]. While currently intended for the transformation of `AutoFOCUS` models, due to its generic nature it supports the transformation of EMF Ecore [14] models in general. As illustrated, the implemented plug-in provides tool support both for the definition of transformation and the transformation execution.

4.1 Transformation Definition

From the point of view of the designer of the transformation, its description consists of

- the declaration of the *Prolog clauses forming the rules* of the transformation relation,
- the declaration of the *root clause of the rule set*, and
- the declaration of the *list of parameters* of the root clause including their type (i.e., input model, output model, Boolean, Integer, String, or model entity),

provided in form of an XML file.

Since transformations depend on the structure of the model before and after the transformation, and transformations are intended to be designed at run-time of the `AutoFOCUS` framework, the designer of the model is provided with information on the structure of the conceptual model at runtime. As shown in the left-hand side of Figure 8, for that purpose, a model structure browser describing the term structure is provided as part

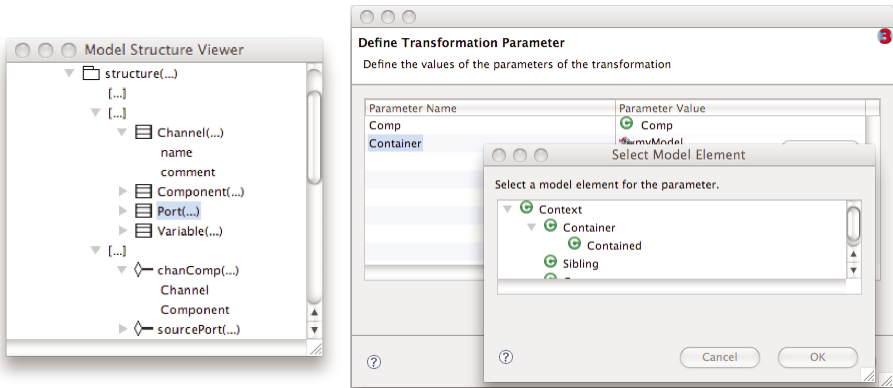


Fig. 8. Support for Defining and Applying Model Transformations

of the online help documentation, based on the EMF Ecore reflection mechanism. By providing the term structure rather than the ECore structure, the declaration of the transformation rules is simplified.

The term structure is represented by providing the tree of relations used to construct the term as shown in Table 1; relations are E.g., as shown in Figure 8, the structure package term can be broken up into lists of subpackage, class, and association terms using the functor `structure`; similarly, the channel element term can be broken up into the attributes `name` and `comment` using the functor `Channel`.

4.2 Transformation Execution

The transformation is provided in form of a transformation wizard, available for Auto-FOCUS projects and guiding the user through the three transformation steps

1. Selecting the transformation declaration
2. Defining the parameters of the transformation
3. Declaring the result model

plus an optional debug step. In the first step, the file containing the transformation description is selected. In the second step, as shown in the right-hand side of Figure 8, the parameters of the transformation rule – without the pre- and post model parameter – are defined. In the case of the push-pull transformation, suitable component entities must be assigned to the parameters *Comp* and *Container* for the to-be-pushed-down component and the container component. To define entity parameters, a model browser is provided to select suitable elements from the pre-model. After declaring, i.e., naming and locating, a model to contain the result of the transformation in the third step, the transformation can be executed. Optionally, the execution can be traced for debugging purposes, to analyse each application of each transformation rule.

The execution of the transformation itself also involves three steps:

1. Translation of the pre-model
2. Application of the transformation relation
3. Translation of the post-model

In the first step, the EMF model is translated into the corresponding representation as a Prolog term as explained in Section 2. Since pre-model construction – just like post-model construction – is of linear complexity concerning the size of the EMF model, its contribution to the overall complexity of a transformation is negligible. Once the term structure of the pre-model is built, the transformation relation is evaluated by the Prolog machine in the next step. Here, the term structure is bound to the pre-model parameter, and the other parameters – except the post-model – are bound to the values selected in the above-mentioned step for the parameter definition. If execution is successful, the resulting model – bound to the post-model parameter – is then translated into the corresponding EMF model and stored as an AutoFOCUS project with the name and location as defined in the above-mentioned step for the result declaration.

5 Conclusion and Outlook

The AutoFOCUS transformation framework supports the transformation of EMF Ecore models using a declarative relational style. While the approach was demonstrated using the AutoFOCUS language for the description of component architectures, the transformation mechanisms can be applied to EMF Ecore models in general, and thus is also suitable for other domain-specific languages. The framework provides a term-structure representation of the EMF Ecore description and offers a Prolog rule-based interpretation, and thus allows a simple, precise, and modular specification of transformation relations on the problem- rather than the implementation-level.

By taking the operational aspects into consideration, the purely relational declarative form of specification can be tuned to ensure an efficient execution. Naturally, the abstraction from the object lattice of an EMF Ecore model to a model term of the conceptual model leads to some loss of efficiency – with the dereferencing of unique associations as the most prominent case (constant vs. linear complexity). However, in our experiments, the approach is practically feasible for real-world sized models (e.g., refactoring models consisting of more than 3000 elements and more than 5000 relations within a few seconds).

While the implementation has demonstrated the applicability of the approach, for a more thorough analysis a larger comparative case study considering the approaches mentioned in Subsection 1.1 is necessary, especially considered usability aspects like complexity of transformation specification and efficiency of executions. Furthermore, since the rule-based approach allows very general forms of application, other forms of application (e.g., view generation, transformations involving user interactions) will be integrated to extend the current implementation.

Finally, the current version of the framework covered those features of EMF that are sufficient to handle the complete AutoFOCUS conceptual model (including, e.g., data types and state machines). For other meta models like the UML 2.1 additional features

(sorted) multi-valued attributes must be included, which – due to the structure- and list-based formalization – can be readily integrated in the current formalization.

Acknowledgements

It is a pleasure to thank Florian Hölzl, Benjamin Hummel, and Elmar Jürgens for their help in getting the implementation running; Florian Hölzl and Peter Braun for the discussion on rule-based transformation; and finally the reviewers for their help in improving the presentation of the approach.

References

- [1] Fowler, M., Scott, K.: UML Distilled. Addison-Wesley, Reading (1997)
- [2] Klar, F., Königs, A., Schürr, A.: Model transformation in the large. In: ESEC/FSE 2007. ACM Press, New York (2007)
- [3] Varró, D., Pataricza, A.: Generic and meta-transformations for model transformation engineering. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 290–304. Springer, Heidelberg (2004)
- [4] Grunske, L., Geiger, L., Lawley, M.: A graphical specification of model transformations with triple graph grammars. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 284–298. Springer, Heidelberg (2005)
- [5] Sprinkle, J., Agrawal, A., Levendovszky, T.: Domain Model Translation Using Graph Transformations. In: ECBS 2003 - Engineering of Computer-Based Systems (2003)
- [6] Rozenberg, G. (ed.): Handbook on Graph Grammars and Computing by Graph Transformation: Foundations. World Scientific, Singapore (1997)
- [7] OMG: Initial submission to the MOF 2.0 Q/V/T RFP. Technical Report ad/03-03-27, Object Management Group (OMG) (2003), <http://www.omg.org>
- [8] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: ATL: a QVT-like transformation language. In: OOPSLA 2006, pp. 719–720. ACM Press, New York (2006)
- [9] Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The Missing Link of MDA. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505. Springer, Heidelberg (2002)
- [10] Lawley, M., Steel, J.: Practical declarative model transformation with tefkat. In: Bruehl, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 139–150. Springer, Heidelberg (2006)
- [11] White, J., Schmidt, D.C., Nechypurenko, A., Wuchner, E.: Introduction to the generic eclipse modelling system
- [12] Varró, G., Friedl, K., Varró, D.: Implementing a graph transformation engine in relational databases. Software and Systems Modeling 5 (2006)
- [13] Schätz, B., Huber, F.: Integrating formal description techniques. In: Woodcock, J.C.P., Davies, J., Wing, J.M. (eds.) FM 1999. LNCS, vol. 1709, p. 1206. Springer, Heidelberg (1999)
- [14] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison Wesley Professional, Reading (2007)
- [15] Jones, S.P.: Haskell 98 language and libraries: the Revised Report. Cambridge University Press, Cambridge (2003)
- [16] Denti, E., Omicini, A., Ricci, A.: Multi-paradigm Java-Prolog integration in tuProlog. Science of Computer Programming 57(2), 217–250 (2005)

A Practical Evaluation of Using TXL for Model Transformation^{*}

Hongzhi Liang and Juergen Dingel

School of Computing, Queen's University, Canada
{liang, dingel}@cs.queensu.ca

Abstract. As one of the MDA's main principles, model transformation has led to the specification of QVT and a large number of model transformation tools. TXL is a generic source transformation tool that also possesses some very important model transformation properties, such as scalability and efficiency. In this paper, we consider TXL as a model transformation tool, especially, for model-to-model transformations. We first present an approach for applying TXL as a model transformation tool. An interesting feature of the approach is the automatic generation of TXL grammars from meta-models. Then, practical applications of model transformation via TXL that follow our approach are given. We conclude the paper with a preliminary evaluation of using TXL as a model transformation tool.

1 Introduction

One of the OMG's Model-driven architecture (MDA) [10] main principles is that system development can be viewed as a process creating a sequence of models. A model can highlight a particular perspective, or view, of the system under development. Model transformation then converts one view to a different view. The target view may be on an equivalent level of abstraction (e.g., from one structural view to another), or it may be on a different level (e.g., from a platform-independent model (PIM) to a platform-specific model (PSM)). To support the principle, the OMG has issued a Request For Proposals (RFP) on a specification of Query/Views/Transformations (QVT) [14]. Either as explicit answers to the request or motivated by practical needs, numerous approaches have been proposed in the literature recently. In [2], Czarnecki and Helsen have presented a set of model transformation comparison criteria, and have provided a survey and categorization of several approaches.

Our interest in model transformation was initiated by our efforts to realize and implement more comprehensive support for the manipulation of requirements in the form of UML Sequence Diagrams [12]. In [20], a general technique to merge Sequence Diagrams is presented. To implement this technique, Sequence Diagrams contained in serialized UML models must be transformed to an internal

^{*} This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Ontario Centres of Excellence (OCE), and IBM CAS Ottawa.

representation called Higher-Order Graphs, then the Higher-Order Graphs are merged and transformed back to Sequence Diagrams.

A large body of work and tools for model transformation already exists [9,30,19,18,26,33,25,27]. However, we decided to employ the source transformation system TXL [1] for this purpose. Given that TXL was not originally designed for model transformation, this may appear as a surprising choice. Our decision was motivated by the following considerations: On the one hand, we felt that our considerable experience with TXL would outweigh the advantages that the use of other approaches might bring. On the other hand, we wanted to study the differences between source and model transformation more closely and see to what extent source transformation can be employed for model transformation. The central purpose of this paper thus is not to suggest that TXL should be used for model transformation in general; rather, we want to study the exact circumstances and conditions under which TXL could be used for this task and present an approach to model transformation, which, under certain circumstances, would allow for existing expertise on TXL in particular and source code transformation in general to be leveraged. In short, a TXL-based model transformation tool would have the following characteristics:

- *Versatility*: In [22], an example of a transformation from a profile of UML models to Java code has already shown that TXL is capable of model-to-code (M2C) transformation. In this paper, we will provide evidence that TXL is capable of model-to-model (M2M) transformation as well. Thus, both M2M and M2C transformations required in the entire MDA “life cycle” could be supported by TXL.
- *Maturity and Scalability*: In the last twenty years, TXL has been widely and successfully used in industry and academia for various software engineering, programming language processing and analysis tasks. Moreover, TXL has been proven to be able to handle very large input files with up to 100,000 source lines per input file. Considering the complexity and scale of modern systems, such maturity and scalability are definitely desired.
- *Efficiency*: As pointed out in [2], the performance of a transformation approach is influenced by various design and implementation strategies, such as *rule application scoping*, *rule application strategy*, and *rule scheduling*. To achieve efficient transformation, TXL allows very precise and flexible control over scoping, rule application, and scheduling strategies.
- *Formal semantics*: In [21], a denotational semantics was given for the TXL programming language. By having a formal semantics, formal reasoning and verification of the correctness of TXL transformation becomes possible, e.g., by automated theorem proving.
- *Gradual learning curve*: In our experience, people with some knowledge in EBNF, and functional and logic programming are able to start programming in TXL fairly easily. Thus, no extensive background in MDA, Object Constraint Language (OCL) [15] and etc. are required to develop TXL rules and grammars for model transformation.

A first evaluation of TXL as a model transformation tool has been performed in [22]. Our paper thus extends this evaluation. More specifically, we would like to show that TXL is capable of both M2C and M2M transformations. We will describe an approach suitable for using TXL as a model transformation tool. Then, we will highlight the advantages and disadvantages of the approach.

The paper is organized as follows. In Section 2, a brief overview of TXL as a generic source transformation tool is provided. Section 3 first identifies the connections and gaps between model transformation and source transformation, and then presents an approach suitable for turning TXL into a model transformation tool. Examples for using TXL as a model transformation tool are given in Section 4. Section 5 discusses the advantages and limitations of the approach and then explores some possible future work. We discuss the related work in Section 6. Finally we conclude the paper in Section 7.

2 Source Transformation via TXL

TXL is a hybrid functional and rule-based programming language particularly designed to support software analysis and source transformation tasks. The overall input-output behavior of TXL transformation is illustrated in Fig. 1. Each TXL program contains two components. First, the syntactic structure of the artifact to be transformed is described by a context-free, possibly ambiguous grammar in EBNF. Second, a rooted set of structural transforming rules has to be given in the form of pattern-replacement pairs. In TXL, rules not only specify rewriting, but also provide the strategy for applying them. The explicitly scoped application of parameterized subrules is scheduled by an implicit top-down search strategy automatically inferred from the rules. The formal semantics and implementation of TXL are based on tree rewriting in which matching transformation rules are applied to the input until a fixed point is reached.

Similar to nonterminal definitions in EBNF, *define* statements (e.g., lines 31 to 36 in the sample grammar in Fig. 2) in TXL grammars give definitions for TXL nonterminals (e.g., `FamilyMMModel_member_Atts`) with nonterminals referenced in square brackets (e.g., `[attvalue]`) and terminal symbols. Please note, in

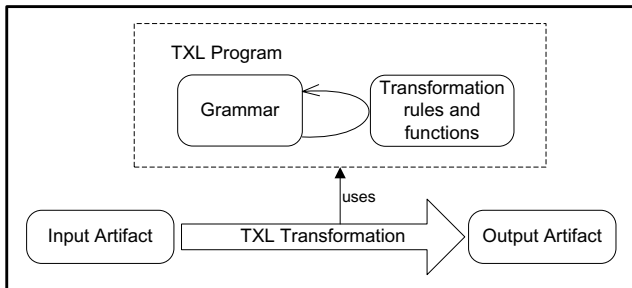


Fig. 1. An overview of TXL processor

```

%%% family.grammar %%%
1 include "XML.Grammar"
2 include "XML.Grammar"
3 include "Ecore.Grammar"

4 define program
5 [prolog] [top_FamilyMModel_element]
6 end define

7 define top_FamilyMModel_element
8 [FamilyMModel_Family]
9 end define

10 define FamilyMModel_Family
11 '<FamilyMModel:Family [repeat FamilyMModel_Family_Atts] '>
12 | '<FamilyMModel:Family [repeat FamilyMModel_Family_Atts] '>
13 [repeat FamilyMModel_Family_Elms]
14 '</FamilyMModel:Family>
15 end define

16 define FamilyMModel_Family_Atts
17 'lastName = [attvalue]
18 | [StartAttrib] | [xmi_attribute_id]
19 | [xmi_attributeGroup_ObjectAttribs]
20 end define

21 define FamilyMModel_Family_Elms
22 [FamilyMModel_Family_member]
23 | [xmi_Extension]
24 end define

25 define FamilyMModel_Family_member
26 '<member [repeat FamilyMModel_member_Atts] '>
27 | '<member [repeat FamilyMModel_member_Atts] '>
28 [repeat FamilyMModel_member_Elms]
29 '</member>
30 end define

31 define FamilyMModel_member_Atts
32 'firstName = [attvalue]
33 | 'relation = [attvalue]
34 | [StartAttrib] | [xmi_attribute_id]
35 | [xmi_attributeGroup_ObjectAttribs]
36 end define

37 define FamilyMModel_member_Elms
38 [xmi_Extension]
39 end define

```

Fig. 2. The grammar of an example TXL program

TXL terminal symbols are either prefixed with a single quote (e.g., 'firstName in line 32 in Fig. 2) or represented directly (e.g., the equal symbol immediately follows 'firstName in line 32). Previously defined nonterminals can also be overridden by using *redefine* statements. Moreover, modularity can be achieved by using *include* statements (e.g., the inclusion of the previously defined XML grammar in line 1 in Fig. 2).

Transformation rules are defined by *rule* (e.g., lines 10 to 22 in Fig. 3) and *function* (e.g., lines 2 to 9 in Fig. 3) statements. The only difference between rules and functions is that a rule searches its pattern on the entire tree it is applied to and replaces every match with its replacement, whereas a function only replaces the first match. Inside a rule, the pattern, e.g., lines 3 and 4 (or replacement, e.g., lines 7 and 8) is defined by a sequence of terminals and variables in the *replace* (or *by*) statement. A rule's pattern and replacement can be further restricted by deconstructors (*deconstruct* statements, e.g., lines 13 and 14), constructors (*construct* statements, e.g., lines 18 and 19) and conditions (*where* statements, e.g., lines 16 and 17). Local variables (either explicitly introduced by *construct* statements, by patterns, or by formal parameters of rules), or global variables (introduced by *import* and *export* statements) (e.g., lines 5, 6 and 15) can also be used to construct rules.

In TXL, rules are explicitly scheduled by the rule calling mechanism. A calling rule directly invokes a sequence of (possibly parameterized) subrules with a given scope, i.e., the input tree of the called subrules. The scope of the applied subrules can be effectively limited by restricting the input tree to particular trees. For instance, it is possible to identify a subtree that matches some predefined pattern by the calling rule, and then the called subrules are applied to the matched subtree rather than the entire input tree of the calling rule. To supplement the default rule application strategy (rules to be applied recursively in order to find every match, and functions to be applied exactly once for the first match), TXL

```

%%% family.txl %%%
1 include "family.grammar"

2 function main
3   replace [program]
4     p [program]
5   export newLastStr[stringlit]
6     _ [quote Simpson]
7   by
8     p [changeLastName]
9 end function

10 rule changeLastName
11   replace $ [FamilyMModel_Family_Atts]
12     att [FamilyMModel_Family_Atts]
13   deconstruct att
14     'lastName= oldLastStr [stringlit]
15   import newLastStr[stringlit]
16   where
17     oldLastStr [~= newLastStr]
18   construct newLastName [attvalue]
19     newLastStr
20   by
21     'lastName = newLastName
22 end rule

```

Fig. 3. The rules of an example TXL program

also supports the so-called one-pass rules. For this kind of rule (specified by a `$` followed by the keyword *replace*), it is applied to each matching item exactly once. The combination of all these mechanisms allows very precise control over the execution of the TXL program and thus presents the experienced TXL user with a very effective means to improve performance and achieve rapid translation even of very large inputs.

For example, a TXL program is shown in Fig. 2 and Fig. 3. The grammar part of the TXL program, which parses XMI files conforming to a specific schema, is given in Fig. 2. When the `main` function in Fig. 3 is invoked, it will call the `changeLastName` rule with the whole input program `p` as the scope. Then, the called `changeLastName` rule will replace a family’s last name to “Simpson” as long as the last name is not equal to “Simpson”.

3 Model Transformation via TXL

A three-layer meta-modeling architecture of our transformation approach is shown in Fig. 4. It is influenced by the architectures used in MOF [11] and EMF [6]. At the top M3 layer is the meta-meta model, e.g., *Ecore* as shown in the diagram. It is used to build meta-models at the middle M2 layer, e.g., MM_s and MM_t . The models, e.g., M_s and M_t conforming to MM_s and MM_t respectively, are at the bottom M1 layer. A TXL model transformation, e.g., the (green)¹ transformation link MT_{st} between M_s and M_t using the TXL program P_{st} inside the (green) dashed box, would take a source model and produce a target model.

However, there is a gap between source transformation and model transformation. The input and output of both transformations seem quite different. On the one hand, source transformation takes a piece of text conforming to a particular grammar as input, e.g., Java source code, and produces another piece of text conforming to another grammar, e.g., C++ source code. On the other hand,

¹ Shown as green on a color display.

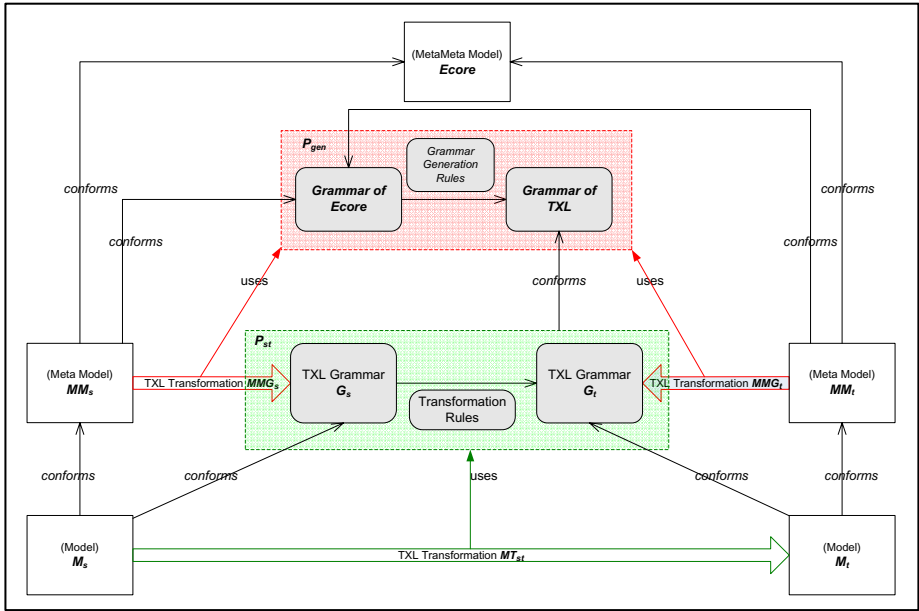


Fig. 4. An overview of model transformation via TXL

the input and output of model transformation are models conforming to meta-models, i.e., presumably diagrams generated by modeling tools. Fortunately, this difference can be overcome because MOF or EMF models, and their conforming meta-models, can be easily serialized as XMI [13] files. As a consequence, a (serialized) model transformation is nothing but a special case of source transformation that takes a source model in one XMI file and produces a target model in another XMI file.

Another factor complicating model transformation via TXL slightly is that the grammars for both input and output artifacts of a TXL program must be combined into a single grammar, which appears to conflict with the fact that the source and target models may conform to different meta-models, and thus may require different TXL grammars. In [3], union and consume/emit grammars are introduced to address tasks that involve different grammars. When input and output artifacts' grammars share a lot of similar concepts, the grammars can be described by a union grammar such that whenever the concepts are matched they are combined. In contrast, when grammars are very different, it is better to combine them by means of a consume/emit grammar where the combination of grammars is only happening at the top level or not at all.

Finally, as a special case of source transformation, grammars for input and output models are needed for each TXL-based (serialized) model transformation. Meta-models (e.g., represented as serialized Ecore meta-models or XMI schemas) are promising starting points, because both meta-models and grammars share a “conforms” relationship with the input/output models. Unfortunately,

meta-models cannot simply be used as grammars directly. Developers of TXL model transformation will have to manually construct a union or a consume/emit grammar by consulting the meta-models for each model transformation. Since (serialized) models are XMI files, one minimal requirement for the resulting grammars is that the grammars must be able to parse XMI files. To achieve that, one can extend the XML grammar [35] with additional XMI elements. In addition to adding XMI elements, when constructing a TXL grammar from a serialized Ecore meta-model, the following Ecore components will be transformed to TXL grammar nonterminals:

- *EPackage*: as the root element of the serialized Ecore meta-model, the *EPackage* will be mapped to a nonterminal that contains a list of all the root elements of the models.
- *EClass*: as an inner element of the meta-model’s root element, i.e., *EPackage*, each *EClass* will be mapped to a nonterminal that specifies a root element of the models. Such nonterminal consists of terminals, and nonterminals of the root element’s inner elements and attributes.
- *EReference* and *EAttribute*: as an inner element of an *EClass*, depending on the lower bound and upper bound values, an *EReference* or *EAttribute* will be mapped to nonterminal(s) that specify inner elements and/or attributes of a root element of the models.

Experienced source code transformation programmers know that the design of a grammar suitable for the transformation task is an important step towards an efficient transformation. Although the manual grammar construction process is relatively simple, it can quickly become tedious and error-prone when creating TXL grammars from complex meta-models, e.g., the meta-model of UML. In contrast, a carefully designed grammar generator can mimic the process of a manual grammar creation process and can generate the same efficient grammars from meta-models. To relieve programmers from this unnecessary burden, we have implemented two automatic TXL grammar generators. One generates grammars from serialized Ecore meta-models (Ecore2Grammar). The Ecore2Grammar generator, shown as P_{gen} in Fig. 4, was implemented as a TXL transformation. To build the Ecore2Grammar, first we created a consume/emit grammar, which combines the grammar of Ecore and the grammar of the TXL language itself at the top level. Then, we created the grammar generation rules, which basically map those Ecore components to TXL nonterminals as described above.² Fig. 2 shows an example of such automatically generated grammar from the serialized Ecore meta-model shown in Fig. 5. TXL grammars generated by Ecore2Grammar ensure model instances are correctly typed according to their meta-models. Constrains, e.g., OCL constraints contained in Ecore meta-models, provide further restrictions on instances models. Currently, the Ecore2Grammar generator does not support OCL constrains. The other generator, XMI2Grammar, is very similar to Ecore2Grammar, except it generates grammars from XMI schemas.

² To integrate with the upcoming Eclipse-based TXL development environment, Ecore2Grammar was implemented in Java as an Eclipse plug-in.

Up to this point, we have all the ingredients we need to let TXL work as a model transformation tool. To develop a model transformation with TXL, we can then follow the general process illustrated in Fig. 4:

1. **Gathering meta-models:** In this step, the meta-models MM_s and MM_t of both input and expected output models are gathered.
2. **Generating grammars:** In this step, input and output models' grammars, G_s and G_t are generated by taking the (red) transformations MMG_s and MMG_t with MM_s and MM_t as inputs, respectively. Both MMG_s and MMG_t transformations could use one of our implemented automatic grammar generators, e.g., TXL program P_{gen} in the (red) dashed-box.
3. **Creating transformation program:** In this step, a TXL program, for instance P_{st} , is created by model transformation developers. The program contains a union or consume/emit grammar which is constructed by combining the grammars generated from the previous step. Transformation rules relating the source and target grammars are also created for the program.
4. **Executing transformation:** In this final step, the TXL program P_{st} created from the last step is executed as the transformation MT_{st} with M_s as the input model and will produce M_t as the output target model.

Finally, the implementation was developed under an Eclipse based TXL development environment.

4 Case Studies

In the previous section, we have identified some of the differences between source transformation and model transformation, and then we have described an approach suitable for using TXL as a model transformation tool. In this section, we illustrate the approach on two examples of model transformation via TXL by following the general process we described in the last section.

4.1 Family2Persons

This example demonstrates the transformation between a **Family** model to a **Persons** model. A slightly different example has also been used by ATL [9].

1. Gathering Meta-models: The meta-model **FamilyMModel** is shown in Fig. 5. It consists of a *Family* class which has a *lastName* attribute, and a *Member* class which has attributes *firstName* and *relation*. A *Family* contains at least one *Member*. A *Member* must belong to a single *Family*, and its *relation* to the *Family* could be, for example, “Father”, “Mother”, “Son”, “Daughter” and so on. The meta-model **PersonsMModel** is shown in Fig. 6. It consists of a *Persons* class, a *Male* class and a *Female* class. A group of *Persons* may contain *Males* and *Females*. Each *Male* or *Female* has a *fullName* attribute.

2. Generating Grammars: From the serialized Ecore meta-models, i.e., **FamilyMModel** and **PersonsMModel** gathered from the last step, two TXL grammars

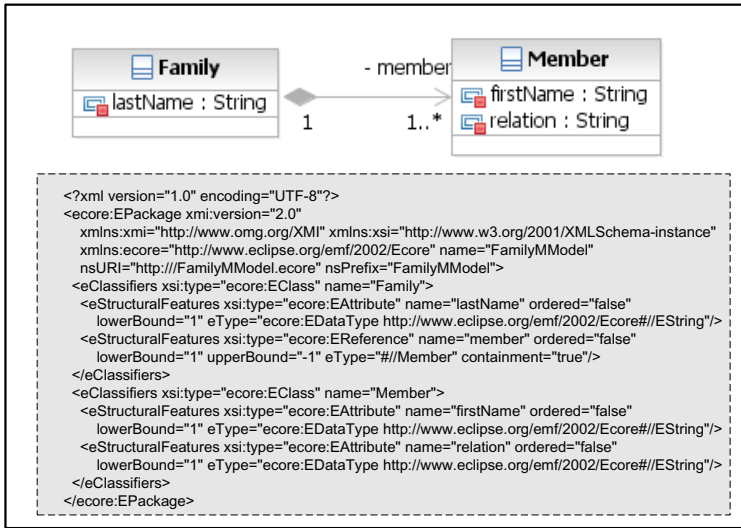


Fig. 5. The FamilyMMModel is shown as a class diagram and a serialized Ecore meta-model

family.grammar and persons.grammar are generated by using the Ecore2Grammar generator. In Sec. 2, family.grammar has already been shown in Fig. 2³. Fig. 7 shows the generated persons.grammar.

3. Creating Transformation Program: Since family.grammar and persons.grammar are quite different, a consume/emit grammar is more suitable than a union grammar. The creation of the consume/emit grammar includes importing the predefined grammars and the generated grammars, and creating nonterminals that combine family.grammar and persons.grammar at the top level. The consume/emit grammar is shown in Fig. 8.

To transform a Family model to a Persons model, the following rules are created and are shown in Fig. 8:

1. *function main* is the required TXL main rule which initiates the transformation by explicitly invoking other rules and functions.
2. *function family2Persons* replaces the input model containing a *Family* by a output model containing a newly constructed *Persons* object with a set of *Males* and *Females*.
3. *function getLastName* extracts the *lastName* from the *Family* and exports it as a global variable.
4. *rule member2MaleOrFemale* constructs either a *Male* or a *Female* for each *Member* of the *Family*. This rule further delegates its constructions to:

³ Actually, the generated grammar doesn't contain the first three include statements and the first define statement. They are included to make the grammar self-contained for the TXL example shown in Sec. 2.

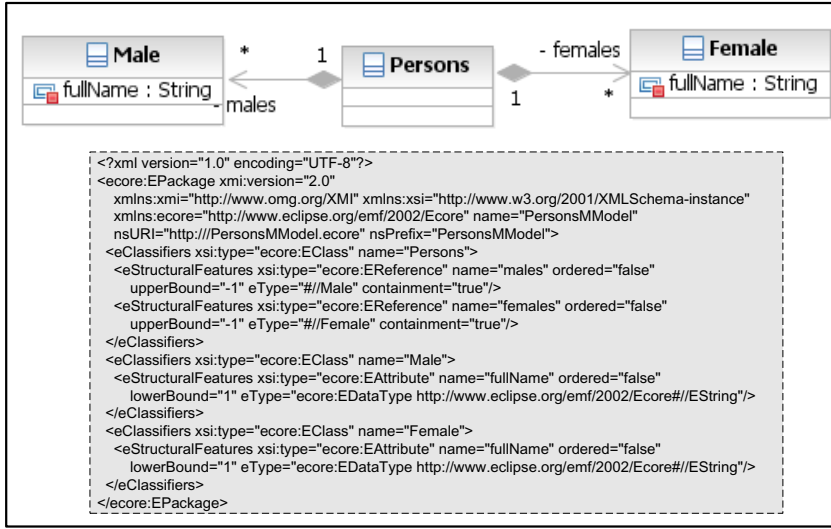


Fig. 6. The PersonsMMModel is shown as a class diagram and a serialized Ecore meta-model

- function* *getFirstName* extracts the *firstName* from a *Member* and exports it as a global variable.
- function* *isMale* first checks whether the *relation* attribute of a *Member* is equal to “Father” or “Son”. If the *relation* is indeed a male relationship, the function will construct a *Male* with a *fullName* constructed by *function* *concatLastName*. The newly constructed *Male* is then added to the set of *Males* and *Females*.
- function* *isFemale* is similar to *function* *isMale* except that it creates a *Female* from a *Member* with either a “Mother” or “Daughter” relation.
- function* *concatLastName* constructs a *Male* or *Female*’s *fullName* by concatenating a *Member*’s *firstName* with the *Family*’s *lastName*.

4. Executing Transformation: When executing the model transformation specified by the TXL program shown in the previous step, a serialized Family model, e.g., the one shown in Fig. 9, is transformed to a serialized Persons model shown in Fig. 10.

4.2 SD2HG and HG2SD

As described in [20], we present an approach to integrating a core subset of UML Sequence Diagrams. In our approach, a Sequence Diagram is represented as a Higher-Order Graph, that is a chain of two type mappings between three graphs, BaseGraph, CollaborationGraph and OccurrenceGraph. Our approach uses category theory as the mathematical foundation and integration is achieved through the explicit representation of the relationships between models via structure-preserving maps and a colimit construction. A prototype Sequence Diagrams

```

define top_PersonsMModel_element
  [PersonsMModel_Persons]
end define

define PersonsMModel_Persons
  '<PersonsMModel:Persons [repeat PersonsMModel_Persons_Atts] />'
  | '<PersonsMModel:Persons [repeat PersonsMModel_Persons_Atts] />'
    [repeat PersonsMModel_Persons_Elms]
  '</PersonsMModel:Persons>'
end define

define PersonsMModel_Persons_Atts
  [StartAttrib] | [xmi_attribute_id] | [xmi_attributeGroup_ObjectAttribs]
end define

define PersonsMModel_Persons_Elms
  [PersonsMModel_Persons_males]
  | [PersonsMModel_Persons_females] | [xmi_Extension]
end define

define PersonsMModel_Persons_males
  '<males [repeat PersonsMModel_Male_Atts] />'
  | '<males [repeat PersonsMModel_Male_Atts] />'
    [repeat PersonsMModel_Male_Elms]
  '</males>'
end define

define PersonsMModel_Persons_females
  '<females [repeat PersonsMModel_Female_Atts] />'
  | '<females [repeat PersonsMModel_Female_Atts] />'
    [repeat PersonsMModel_Female_Elms]
  '</females>'
end define

define PersonsMModel_Male_Atts
  'fullName = [attvalue]'
  | [StartAttrib] | [xmi_attribute_id]
  | [xmi_attributeGroup_ObjectAttribs]
end define

define PersonsMModel_Male_Elms
  [xmi_Extension]
end define

define PersonsMModel_Female_Atts
  'fullName = [attvalue]'
  | [StartAttrib] | [xmi_attribute_id]
  | [xmi_attributeGroup_ObjectAttribs]
end define

define PersonsMModel_Female_Elms
  [xmi_Extension]
end define

```

Fig. 7. The generated persons.grammar

integration tool following our approach has been implemented as an extension to Eclipse. The prototype tool requires two model transformations, i.e., a model transformation from UML Sequence Diagrams to Higher-Order Graphs (SD2HG) and another model transformation on the opposite direction, Higher-Order Graphs to Sequence Diagrams (HG2SD).

Both SD2HG and HG2SD have already been implemented in TXL by following the same general process used in the Family2Persons example. In contrast to the Family2Persons model transformation, which is a toy example, SD2HG and HG2SD are working model transformations motivated by real needs. Due to the space limit, we will only highlight some interesting aspects of the transformations, and particularly focus on the parts that are related to BaseGraphs in SD2HG.

In the process, some relatively large and complex meta-models are involved. Instead of creating a new Ecore meta-model for Sequence Diagrams (such as the one described in [12]), we used the existing serialized Ecore meta-model of UML provided in [7], which contains more than ten thousand lines of text. Then, the meta-model was successfully transformed to the TXL grammar of UML by using the Ecore2Grammar generator. Similarly, the meta-model of Higher-Order Graphs has been transformed to the TXL grammar of Higher-Order Graphs. In particular, a BaseGraph, whose meta-model is shown in Fig. 11, is very similar to a UML Class Diagram, except that the edges represent dynamic rather than static associations. We therefore interpret them as message types. A root node called *Class* is contained in every BaseGraph. For concurrent systems, an object of a class may or may not own a thread of control. In other words, an object can be specified as either an active object or a passive object. To distinguish these two different types of objects in our formalization, we introduced two sub-classes of the root *Class*, *ActiveClass* and *PassiveClass*. We assume that an object of

```

include "XML.Grammar"
include "XML.Grammar"
include "Misc.Grammar"
include "Ecore.Grammar"
include "family.grammar"
include "persons.grammar"

define program
  [FamilyMModel] | [PersonsMModel]
end define

define FamilyMModel
  [prolog] [top_FamilyMModel_element]
end define

define PersonsMModel
  [prolog] [top_PersonsMModel_element]
end define
%% % % % END of consume/emit grammar definition % % % %

function main
  replace [program]
    family [program]
  export persons_elms [repeat PersonsMModel_Persons_Elms]
  by
    family [getLastName] [member2MaleOrFemale] [family2Persons]
end function

function family2Persons
  replace * [program]
    familyModel [FamilyMModel]
  deconstruct familyModel
    pro [prolog] family [FamilyMModel_Family]
  import persons_elms [repeat PersonsMModel_Persons_Elms]
  construct persons [PersonsMModel_Persons]
    <PersonsMModel:Persons xmi:version="2.0"
      xmlns:xmi="http://www.omg.org/XMI"
      xmlns:PersonsMModel="http://PersonsMModel.ecore">
  construct personsModel [PersonsMModel]
    pro persons
  by
    personsModel
end function

function getLastName
  match * [FamilyMModel_Family_Atts]
    'lastName = lastName [stringlit]
  export lastName [stringlit]
end function

rule member2MaleOrFemale
  replace $ [FamilyMModel_Family_member]
    member [FamilyMModel_Family_member]
  where
    member [getFirstName]
  where
    member [isMale] [isFemale]
  by
    member
end rule

function getFirstName
  match * [FamilyMModel_member_Atts]
    'firstName = firstName [stringlit]
  export firstName
end function

function isMale
  match * [FamilyMModel_member_Atts]
    'relation = relationStr [stringlit]
  construct FatherStr [stringlit]
    _ [+ "Father"]
  construct SonStr [stringlit]
    _ [+ "Son"]
  where
    relationStr [= FatherStr] [= SonStr]
  import firstName [stringlit]
  where
    firstName [concatLastName]
  import fullName [stringlit]
  construct maleNameAtt [PersonsMModel_Male_Atts]
    'fullName = fullName
  construct newMale [PersonsMModel_Persons_Elms]
    < 'males maleNameAtt />
  import persons_elms [repeat PersonsMModel_Persons_Elms]
  export persons_elms
    persons_elms [ newMale]
end function

function isFemale
  match * [FamilyMModel_member_Atts]
    'relation = relationStr [stringlit]
  construct MotherStr [stringlit]
    _ [+ "Mother"]
  construct DaughterStr [stringlit]
    _ [+ "Daughter"]
  where
    relationStr [= MotherStr] [= DaughterStr]
  import firstName [stringlit]
  where
    firstName [concatLastName]
  import fullName [stringlit]
  construct femaleNameAtt [PersonsMModel_Female_Atts]
    'fullName = fullName
  construct newFemale [PersonsMModel_Persons_Elms]
    < 'females femaleNameAtt />
  import persons_elms [repeat PersonsMModel_Persons_Elms]
  export persons_elms
    persons_elms [ newFemale]
end function

function concatLastName
  match * [stringlit]
    firstName [stringlit]
  import lastName [stringlit]
  construct fullName [stringlit]
    firstName [+ " "] [+ lastName]
  export fullName
end function

```

Fig. 8. The TXL program for family to persons model transformation

ActiveClass can be in one of two states: “executing” and “blocked and waiting for a response”. To capture this, we introduce two special message types *exec* and *wait*. Similarly, *exec* and *rest* are introduced to capture the states of “executing” and “ready and really doing nothing” of an object of *PassiveClass*.

To transform Sequence Diagrams to Higher-Order Graphs, over a hundred TXL rules and functions has been developed for both SD2HG and HG2SD with combined more than four thousand lines of code. Particularly, to generate a BaseGraph from a Sequence Diagram, the following three types of rules have been implemented.

1. Application scoping: A few of the rules were purely implemented for the purpose of limiting application scoping of other rules. For instance, when

```
<?xml version="1.0" encoding="UTF-8"?>
<FamilyMModel:Family xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:FamilyMModel="http://FamilyMModel.ecore" lastName="Simpson">
  <member firstName="Homer" relation="Father"/>
  <member firstName="Marge" relation="Mother"/>
  <member firstName="Bart" relation="Son"/>
  <member firstName="Lisa" relation="Daughter"/>
  <member firstName="Maggie" relation="Daughter"/>
</FamilyMModel:Family>
```

Fig. 9. An example of input *Family* model

```
<?xml version="1.0" encoding="UTF-8"?>
<PersonsMModel:Persons xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:PersonsMModel="http://PersonsMModel.ecore">
  <males fullName="Homer Simpson"/>
  <females fullName="Marge Simpson"/>
  <males fullName="Bart Simpson"/>
  <females fullName="Lisa Simpson"/>
  <females fullName="Maggie Simpson"/>
</PersonsMModel:Persons>
```

Fig. 10. The output *Persons* model

applied with an input UML model, the rules *uml2HGs* and *collaboration2HG* in Fig. 12 will limit the application scope of further called rules from UML Model to UML Collaboration and then to UML Interaction.

2. Control and scheduling: Some of the rules were created for the purposes of control and scheduling of other rules. For example, the rule *interaction2HG* in Fig. 12 first declares and exports three global variables that are used to store intermediate transformation results, and then explicitly invokes other defined rules in sequence.
3. Transformation: A majority of the rules were implemented to either generate target model elements or produce intermediate results. The rule *message2MsgTypeTmp* in Fig. 13, for example, will produce one *MsgTypeTmp* (a temporary representation of *MsgType*) from each *Message* in a UML Interaction. Its tasks consist of 1) collecting the source and target classes associated with the two *MessageEnds*, i.e., the receive and send events of the *Message*, 2) gathering the name of the *MsgType*, and 3) constructing the *MsgTypeTmp* and storing it to the predefined global variable *MsgTypeTmps*. The tasks are then further delegated to other supporting rules.

Although the first two kinds of rules do not produce any target model elements, they are vitally important for more efficient model transformations and to ensure the generation of desired target models.

Finally, the implemented SD2HG and HG2SD have been extensively tested with our prototype Sequence Diagrams integration tool. Serialized UML2 models containing Sequence Diagrams, which were generated by IBM Rational Software Architect (RSA)[16], have been used as inputs to SD2HG. Output models of SD2HG, i.e., serialized Higher-Order Graphs, have been successfully generated

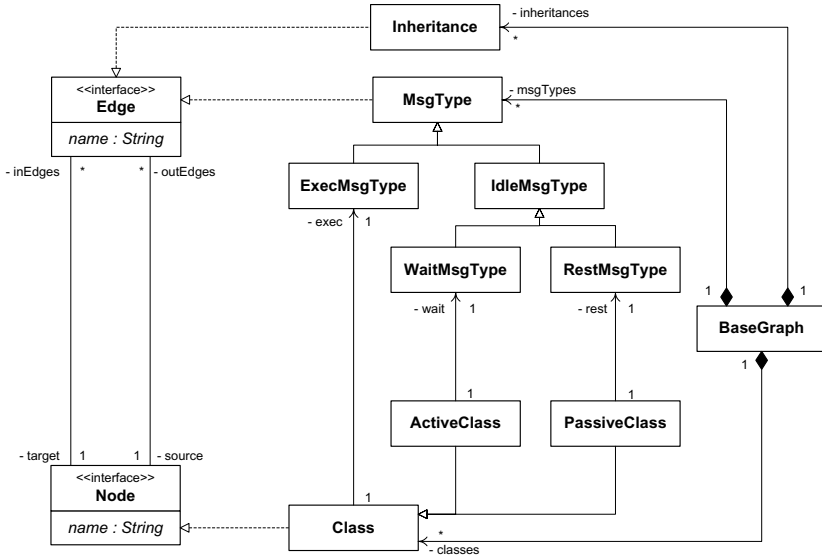


Fig. 11. A partial meta-model of Higher-Order Graphs

and displayed by a graphical editor generated by GMF [5]. Similarly, Higher-Order Graphs have been transformed back to UML2 models containing Sequence Diagrams by HG2SD. The UML2 modes were then visualized by RSA.

The process of implementing SD2HG and HG2SD transformations gave us the opportunity to explore model transformation via TXL more thoroughly. The advantages and the limitations that affect the practicality of model transformation via TXL and some possible future work have been identified and will be discussed in Sec. 5.

```

rule uml2HG model [program]
  replace $ [uml_Package_packageElement]
  collaboration [uml_Package_packageElement]
  where
    collaboration [isCollaboration]
  by
    collaboration [collaboration2HG model collaboration]
end rule
function isCollaboration
  match * [xmi_attributeGroup_ObjectAttribs]
  xmi:type = "uml:Collaboration"
end function

rule collaboration2HG model [program]
  collaboration [uml_Package_packageElement]
  replace $ [uml_BehavioredClassifier_ownedBehavior]
  interaction [uml_BehavioredClassifier_ownedBehavior]
  where
    interaction [isInteraction]
  by
    interaction [interaction2HG model collaboration]
end rule
function isInteraction
  match * [xmi_attributeGroup_ObjectAttribs]
  xmi:type = "uml:Interaction"
end function

rule interaction2HG model [program]
  collaboration [uml_Package_packageElement]
  replace $ [uml_BehavioredClassifier_ownedBehavior]
  interaction [uml_BehavioredClassifier_ownedBehavior]
  export basegraph_Elms [repeat higherordergraph_BaseGraph_Elms]
  export MsgTypeTmps [repeat MsgTypeTmp]
  export ClassTmps [repeat ClassTmp]
  by
    interaction [message2MsgTypeTmp model collaboration interaction]
    [msgTypeTmps2ClassTmps model]
    [lifeline2ClassTmps model collaboration]
    [buildBasegraph]
end rule

```

Fig. 12. Scoping and scheduling rules

```

rule message2MsgTypeTmp model [program] collaboration [uml_Package_packageElement] interaction [uml_BehavedClassifier_ownedBehavior]
  replace $ [uml_Interaction_message]
    msg [uml_Interaction_message]

    %%%% handle the receive event
    %get the "receiveEvent" and the target class of the message
    where
      msg [getReceiveEvent]
      import rtnGetReceiveEvent [id]
      %get the class for the lifeline that contain the receive event
      where
        interaction [getClassByEvent model collaboration rtnGetReceiveEvent]
      import rtnGetClassByEvent [uml_Package_packageElement]
      construct targetClass [uml_Package_packageElement]
      rtnGetClassByEvent
      ..... %handle the send event, similar to receive event

    %%%%Get the name of the message type
    where
      %get the op name from the send event
      interaction [getOpNameByEvent rtnGetSendEvent]
      import returnGetOpName [id]
      construct opName [id]
      returnGetOpName

    %%%%construct the msg type
    %get xmi:id of the target class
    where
      targetClass [getXmiid]
      import returnGetXmiid [id]
      construct targetClassXmiid [id]
      returnGetXmiid
      ..... %get xmi:id of the source class
      construct msgTypeXmiid [id]
      opName
      construct tmpMsgType [MsgTypeTmp]
      opName sourceClassXmiid targetClassXmiid msgTypeXmiid
      import MsgTypeTmps [repeat MsgTypeTmp]
      export MsgTypeTmps
      MsgTypeTmps [addMsgTypeTmpToList tmpMsgType]
    by
      msg
  end rule

```

Fig. 13. A transformation rule

5 Advantages, Limitations and Future Work

As a model transformation tool, TXL has some strong advantages. Some advantages, such as scalability and efficiency, are directly coming from the characteristics that TXL has. There are also other advantages we discovered during our implementation of SD2HG and HG2SD, which are worth mentioning.

- TXL allows fine-grained control over the target model construction. For instance, the details of the placement of a model element inside a model can easily be controlled via the scheduling of TXL rules, and manipulation of global variables. In many situations the additional effort for writing these rules is outweighed by the benefits of such fine-grained control. This fine-grained control is especially helpful in our case as we were dealing with ordered sets of model elements. For instance, when manipulating Sequence Diagrams, it is of vital importance to maintain the order of event and message occurrences across transformations.
- In addition to the nonterminal types in the grammars that are generated from meta-models, TXL allows the rapid creation of new nonterminals. More precisely, new nonterminal types representing intermediate transformation results can be added to the grammars without the need to modify the original meta-models. Paired with global variables, intermediate results are processed in an efficient way yet still benefit from the strongly-typed characteristic of TXL language.

TXL also has some limitations. Some of them are related to the use of TXL in general.

- TXL lacks a flexible rule/function return mechanism. As a functional language, only a single return, i.e., the replacement, is allowed. Moreover, the type of the return value is bound to the type of the pattern of the same rule/function, which could be fairly restrictive as a desired return value may

have a different type than that of the pattern. To relax this restriction, a large number of TXL global variables creations and accesses were used in our SD2HG transformation. Although this approach fulfilled our needs for return values with different types, it did introduce more complexity into our SD2HG and HG2SD implementations.

- TXL does not have “if clauses”. Consequently, whenever branching is needed in a rule, the rule has to be duplicated up to the branching point. The rule duplications not only increased the size, but also negatively affected testability and maintainability of our implementation.

Both limitations were also identified in [29], and suitable solutions were given to address them as part of new version of TXL - ETXL. The following limitation applies to serialized model transformation specifically.

- As explained in Sec. 3, TXL currently is only applicable as a model transformation tool when both models and meta-models can be serialized. This could be inconvenient for users as explicit export operations may be needed to obtain serialized input models from modeling tools, and import operations may also be required before further manipulations on output models can be carried out.

The above limitation suggests some interesting future work. It might be resolved by properly integrating the upcoming Eclipse-based TXL IDE with Eclipse-based modeling tools such that explicit model import/export operations become unnecessary. Other possible topics we would like to pursue as future work are:

- As mentioned earlier, our Ecore2Grammar generator currently does not support OCL constraints. However, we would like to investigate the possibility to generate TXL rules from OCL constraints. The concept of generating rules from OCL constraints has been discussed in [34], where a restricted form of OCL constraints can be translated to graph constraints. Such generated rules then can be utilized to perform well-formedness checks on models.
- The built-in tracing as part of TXL debug options shows the sequence of rules applied during a transformation. Although TXL does not support traceability between target model and source model directly, i.e., the links between target model elements and source model elements may be scattered and interleaved throughout the applied rules, traceability could be established by reorganizing the tracing information provided by TXL in a more human readable format.
- In Sec. 3, we provided automatic grammar generators. It would be desirable if the rules that operate on these generated grammars could be generated as well, even as skeleton rules or functions. We are interested in the following two different approaches. For the first approach, we could fuse a graphical rule editor with TXL. More precisely, we can take advantage of graphical rule specifications provided by some of the graph transformation approaches and then translate graphical rules to TXL rules. This hybrid approach could be further used to realize the generic model transformation based on category

theory's pull-back operation explained in [4]. In the second approach, we could follow the "rule by examples" principle established in TXL programming and use heuristics to infer (possibly partial) rules from user provided transformation examples.

6 Related Work

Similar to our TXL grammar generators, generation of grammars from meta-models is one of the main features of TCS (Textual Concrete Syntax) [17]. Through transformations, associated meta-model elements, which are specified in a user defined TCS model, are translated to grammar elements. TCS supports a broader range of capabilities, e.g., text-to-model and model-to-text transformations. In contrast, our grammar generators are tailor made for the automatic generation of TXL grammars from Ecore meta-models or XMI schemas without user intervention.

Paige and Radjenovic have used TXL as model transformation tool in [22]. Clearly, our work is closely related to theirs. In [22], connecting MOF and TXL to allow TXL specifications to be automatically generated, has been identified as fruitful future work. Our work established exactly this link through the implementation of automatic grammar generators.

The process and techniques used in this paper could also be applied to other source transformation tools. XSLT [32], for example, is a source transformation tool that specifically targets the transformation between XML documents. Since XMI documents are also XML documents, XSLT is also capable of model transformation. That capability has already been demonstrated by various XSLT-based model transformation approaches, e.g., UMT [8] and MTRANS [23]. However, TXL-based approaches, such as ours, are applicable to a broader class of model transformations, because XSLT is not suitable for M2C transformation.

Since we applied TXL as a model transformation tool, this work is inevitably related to tools that have a direct goal of model transformation. Roughly speaking, the model transformation tools can be grouped into two categories, relational-based and graph-transformation-based.

On the one hand, our approach shares many similarities and is much closer to relational-based approaches. The relationship between source model and target model is expressed textually following the same "rule", "pattern", "replacement" and "condition" paradigm as in TXL. Some examples include ATL [9], MT [30], Tefkat [19] and Epsilon Transformation Language (ETL) [18]. Similar to our approach, ATL supports the transform of serialized models. Rule scheduling in ATL can be achieved explicitly by invoking called rules inside the imperative code section of another rule. MT is another relational-based transformation language. It comes with predefined top-down rule scheduling. This simple scheduling makes understanding of transformation programs easier. Yet, we feel that it might impose an unnecessary restriction on programmers to author new transformations, and also could be an unnecessary burden for programmers to maintain existing transformation programs. In contrast to our TXL-based approach, both ATL

and MT approaches do not support flexible application scoping on declarative rules. When dealing with complex source models, the lack of a flexible rule application scoping could lead to less efficient model transformations. Moreover, both transformation execution algorithms do not support ordering on target elements creation, or more precisely, no user control over the placement of target model elements. While both approaches provide traceability information in a much mature way, we consider traceability as one of our future work.

On the other hand, models, especially UML models, are visualized as graphs and diagrams. Therefore, graph transformation, as used in, e.g., Progres [28] and AGG [24], is a natural fit in the context of model transformation. Several graph-transformation-based approaches have already been proposed to address the program of model transformation, for instance, GReAT [26], UMLX [33], ATOM³ [25], and VIATRA2 [27]. Unlike our TXL-based approach, the “rule”, “pattern”, “replacement” and “condition” paradigm can be expressed graphically or textually to manipulate graphs.

Most of the graph-transformation-based approaches provide a sophisticated rule scheduling algorithm. For instance, VIATRA2 uses abstract state machines to manipulate the rule scheduling. In GReAT, rule scheduling is supported by a separate control flow language. While the explicit rule calling algorithm used in TXL only allows sequential execution, depending on how they are connected by the control flow language, the rules in GReAT can be scheduled running both in sequential and in parallel. Rule application scoping is supported by both GReAT and VIATRA2 through limiting the search space of a transformation rule to a subgraph of the source graph. Compared to our approach, rule application for both GReAT and VIATRA2 is limited to either all the valid matches or only the first valid match, but not recursively as our TXL-based approach. One distinct feature provided by GReAT is the ability to produce C++ code from transformation rules, which potentially could speed up the transformations by executing the C++ code directly. Also, transformation rules and control flows in GReAT could be specified graphically. As discussed previously, we consider providing a graphical rule editor as a future work.

In comparison to existing model transformation approaches, our approach allows existing TXL expertise to be leveraged and offers fine grained control over the transformation process at the expense of forcing the developers to work on a lower level of abstraction. The differences in other attributes of transformation techniques, such as conciseness and expressiveness of pattern languages, are also affecting the effectiveness, scalability and maintainability of the different model transformation approaches. As the number of model transformation approaches has steadily increased in the recent years, it would be interesting to compare model transformation via TXL with other approaches, such as ATL, MT, ETL and so on, more comprehensively and systematically. Although measuring and quantifying the differences between approaches is likely a complex job, the criteria provided in [2] and [31] should at least provide some good starting points. However, such comparison is not within the scope of this paper.

7 Conclusions

Model transformation, as an important and necessary component of MDA, has led to the proposal of QVT and a large number of model transformation approaches. As an efficient and scalable source transformation tool, TXL has been applied to a variety of different tasks. In this paper, we presented an approach allowing TXL to be used as a model transformation tool. Our approach benefits from explicit rule scoping, flexible rule application strategy and user controllable rule scheduling provided by TXL, which are some of the essential properties that affect the performance and efficiency of model transformations. Even without the support of OCL constraints, our grammar generators significantly ease the task of creating model transformations through the automatic generation of TXL grammars from meta-models. Then, we highlighted the advantages and limitations of model transformation via TXL based on our own experiences of using TXL as a M2M transformation tool.

References

1. Cordy, J.: The TXL Source Transformation Language. *Science of Computer Programming* 61(3) (2006)
2. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture*, Anaheim, CA, USA (October 2003)
3. Dean, T.R., Cordy, J.R., Malton, A.J., Schneider, K.A.: Grammar programming in TXL. In: *2nd Int. Workshop on Source Code Analysis and Manipulation (SCAM 2002)* (2002)
4. Diskin, Z., Dingel, J.: A metamodel-independent framework for model transformation: Towards generic model management patterns in reverse engineering. In: *3rd Int. Workshop on Metamodels, Schemas, Grammars, and Ontologies* (2006)
5. Eclipse Foundation. Eclipse Graphical Modeling Framework (GMF), <http://www.eclipse.org/gmf/>
6. Eclipse Foundation. Eclipse Modeling Framework Project (EMF), <http://www.eclipse.org/modeling/emf/>
7. Eclipse Foundation. Eclipse UML2, <http://www.eclipse.org/modeling/mdt/?project=uml2>
8. Grønmo, R., Oldevik, J.: An empirical study of the UML model transformation tool (UMT). In: *First Interoperability of Enterprise Software and Applications*, Geneva, Switzerland (February 2005)
9. ATLAS Group. Atlas transformation language, <http://www.eclipse.org/m2m/at1/>
10. Object Management Group. MDA guide version 1.0.1, <http://www.omg.org/docs/omg/03-06-01.pdf>
11. Object Management Group. Meta-Object Facility (MOF), version 2.0, <http://www.omg.org/mof/>
12. Object Management Group. Unified Modeling Language, version 2.1.1, <http://www.omg.org/technology/documents/formal/uml.htm>
13. Object Management Group. XML Metadata Interchange (XMI), version 2.1.1, <http://www.omg.org/cgi-bin/doc?formal/2007-12-01>

14. Object Management Group. OMG/RFP/QVT MOF 2.0 Query/Views/ Transformations RFP (2003), <http://www.omg.org/docs/ad/02-04-10.pdf>
15. Object Management Group. Object Constraint Language Specification Version 2.0 (2006), <http://www.omg.org/technology/documents/formal/ocl.htm>
16. IBM. Rational Software Architect, <http://www-306.ibm.com/software/awdtools/architect/swarchitect/>
17. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: 5th International Conference on Generative programming and Component Engineering (GPCE 2006) (2006)
18. Kolovos, D., Paige, R., Polack, F.: Eclipse Development Tools for Epsilon. In: Eclipse Summit Europe, Eclipse Modeling Symposium (2006)
19. Lawley, M., Steel, J.: Practical declarative model transformation with tefkat. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 139–150. Springer, Heidelberg (2006)
20. Liang, H., Diskin, Z., Posse, J.D.E.: A general approach for scenario integration. In: ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008) (October 2008)
21. Malton, A.J.: The denotational semantics of a functional tree-manipulation language. *Computer Languages* 19(3), 157–168 (1993)
22. Paige, R.F., Radjenovic, A.: Towards model transformation with TXL. In: Proc. Metamodelling for MDA Workshop, York, UK (November 2003)
23. Peltier, M., Bézivin, J., Guillaume, G.: MTRANS: A general framework, based on XSLT, for model transformations. In: Proceedings of the Workshop on Transformations in UML (WTUML 2001), Genova, Italy (April 2001)
24. AGG project. AGG system, <http://tfs.cs.tu-berlin.de/agg/>
25. ATOM³ project. ATOM³ A Tool for Multi-formalism Meta-Modelling, <http://atom3.cs.mcgill.ca/>
26. GReAT project. GReAT: Graph Rewriting And Transformation, <http://www.isis.vanderbilt.edu/Projects/mobies/downloads.asp>
27. VIATRA2 project, <http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html/>
28. Schürr, A., Winter, A.J., Zündorf, A.: The Progres approach: Language and environment. In: Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools. World Scientific, Singapore (1997)
29. Thurston, A.D., Cordy, J.R.: Evolving TXL. In: 6th Int. Workshop on Source Code Analysis and Manipulation (SCAM 2006) (2006)
30. Tratt, L.: The MT model transformation language. In: Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 2006), Dijon, France (2006)
31. Varró, G., Schürr, A., Varró, D.: Benchmarking for graph transformation. In: Symposium on Visual Languages and Human-Centric Computing (2005)
32. W3C. XSL Transformations Version 1.0, <http://www.w3.org/TR/xslt>
33. Willink, E.: UMLX: A graphical transformation language for MDA. In: Workshop on Model Driven Architecture: Foundations and Applications (June 2003)
34. Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. *Electron. Notes Theor. Comput. Sci.* 211, 159–170 (2008)
35. Zhang, H., Cordy, J.: XML 1.0. Grammar and Well-formedness Checker, 2nd edn., <http://www.txl.ca/nresources.html>

DEFACTO: Language-Parametric Fact Extraction from Source Code

H.J.S. Basten and P. Klint

Centrum Wiskunde & Informatica, P.O. Box 94079,
NL-1090 GB Amsterdam, The Netherlands
`H.J.S.Basten@cwi.nl`, `P.Klint@cwi.nl`

Abstract. Extracting facts from software source code forms the foundation for any software analysis. Experience shows, however, that extracting facts from programs written in a wide range of programming and application languages is labour-intensive and error-prone. We present DEFACTO, a new technique for fact extraction. It amounts to annotating the context-free grammar of a language of interest with *fact* annotations that describe how to extract elementary facts for language elements such as, for instance, a declaration or use of a variable, a procedure or method call, or control flow statements. Once the elementary facts have been extracted, we use relational techniques to further enrich them and to perform the actual software analysis.

We motivate and describe our approach, sketch a prototype implementation and assess it using various examples. A comparison with other fact extraction methods indicates that our fact extraction descriptions are considerably smaller than those of competing methods.

1 Introduction

A call graph extractor for programs written in the C language extracts (caller, callee) pairs from the C source code. It contains knowledge about the syntax of C (in particular about procedure declarations and procedure calls), and about the desired format of the output pairs. Since call graph extraction is relevant for many programming languages and there are many similar extraction tasks, it is wasteful to implement them over and over again for each language; it is better to take a generic approach in which the language in question and the properties to be extracted are parameters of a generic extraction tool. There are many and diverse applications of such a generic fact extraction tool: ranging from collecting relevant metrics for quality control during development or managing software portfolios to deeper forms of analysis for the purpose of spotting defects, finding security breaches, validating resource allocation, or performing complete software renovations.

A general workflow for *language-parametric* software analysis is shown in Figure 1. Starting point are *Syntax Rules*, *Fact Extraction Rules*, and *Analysis Rules*. *Syntax Rules* describe the syntax of the system or source code to be analyzed. In a typical case this will be the grammar of C, C++, Java or

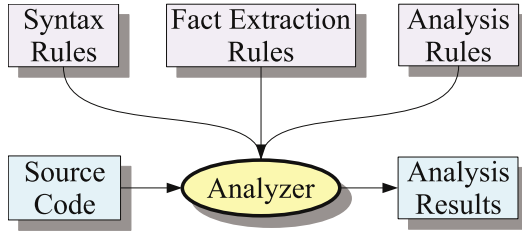


Fig. 1. Global workflow of fact extraction and source code analysis

Cobol possibly combined with the syntax rules for some embedded or application languages. *Fact Extraction Rules* describe what elementary facts have to be extracted from the source code. This may, for example, cover the extraction of variable definitions and uses, and the extraction of the control flow graph. Observe that these extraction rules are closely tied to the context-free grammar and differ per language. *Analysis Rules* describe the actual software analysis to be performed and express the desired operations on the facts, e.g., checking the compatibility of certain source code elements or determining the reachability of a certain part of the code. The *Analyzer* reads the source code and extracts *Facts*, and then produces *Analysis Results* guided by the *Analysis Rules*. Analysis Rules have a weaker link with a programming language and may in some cases even be completely language-agnostic. The analysis of multi-language systems usually requires different sets of fact extraction rules for each language, but only one set of analysis rules.

In this paper we explore the approach just sketched in more detail. The emphasis will be on fact extraction, since experience shows that extracting facts from programs written in a wide range of programming and application languages is labour-intensive and error-prone. Although we will use relational methods for processing facts, the approach as presented here works for other paradigms as well.

The main contributions of this work are an explicit design and prototype implementation of a language-parametric fact extraction method.

1.1 Related Research

Lexical analysis. The mother and father of fact extraction techniques are probably Lex [25], a scanner generator, and AWK [1], a language intended for fact extraction from textual records and report generation. Lex is intended to read a file character-by-character and produce output when certain regular expressions (for identifiers, floating point constants, keywords) are recognized. AWK reads its input line-by-line and regular expression matches are applied to each line to extract facts. User-defined actions (in particular print statements) can be associated with each successful match. This approach based on regular expressions is in wide use for solving many problems such as data collection, data mining, fact extraction, consistency checking, and system administration. This same approach

is used in languages like Perl, Python, and Ruby. The regular expressions used in an actual analysis are language-dependent. Although the lexical approach works very well for ad hoc tasks, it cannot deal with nested language constructs and in the long turn, lexical extractor become a maintenance burden.

Murphy and Notkin have specialized the AWK-approach for the domain of fact extraction from source code [30]. The key idea is to extend the expressivity of regular expressions by adding context information, in such a way that, for instance, the begin and end of a procedure declaration can be recognized. This approach has, for instance, been used for call graph extraction [31] but becomes cumbersome when more complex context information has to be taken into account such as scope information, variable qualification, or nested language constructs. This suggests using grammar-based approaches.

Compiler instrumentation. Another line of research is the explicit instrumentation of existing compilers with fact extraction capabilities. Examples are: the GNU C compiler GCC [13], the CPPX C++ compiler [5], and the Columbus C/C++ analysis framework [12]. The Rigi system [29] provides several fixed fact extractors for a number of languages. The extracted facts are represented as tuples (see below). The CodeSurfer [14] source code analysis tool extracts a standard collection of facts that can be further analyzed with built-in tools or user-defined programs written in Scheme. In all these cases the programming language as well as the set of extracted facts are fixed thus limiting the range of problems that can be solved.

Grammar-based approaches. A more general approach is to instrument the grammar of a language of interest with fact extraction directives and to automatically generate a fact extractor. This generator-based approach is supported by tools like Yacc, ANTLR, ASF+SDF Meta-Environment, and various attribute grammar systems [20,33,10]. Our approach is an extension of the Syntax Definition Formalism SDF [16] and has been implemented as part of the ASF+SDF Meta-Environment [4]. Its fact extraction can be seen as a very light-weight attribute grammar system that only uses synthesized attributes. In attribute grammar systems the further processing of facts is done using attribute equations that define the values of synthesized and inherited attributes. Elementary facts can be described by synthesized attributes and are propagated through the syntax tree using inherited attributes. Analysis results are ultimately obtained as synthesized attributes of the root of the syntax tree. In our case, the further processing of elementary facts is done by using relational techniques.

Queries and Relations. Although extracted facts can be processed with many computational techniques, we focus here on relational techniques. Relational processing of extracted facts has a long history. A unifying view is to consider the syntax tree itself as “facts” and to represent it as a relation. This idea is already quite old. For instance, Linton [27] proposes to represent all syntactic as well as semantic aspects of a program as relations and to use SQL to query them. He encountered two large problems: the lack of expressiveness of SQL

(notably the lack of transitive closures) and poor performance. Recent investigations [3,15] into efficient evaluation of relational query languages show more promising results.

In Rigi [29], a tuple format (RSF) is introduced to represent relations and a language (RCL) to manipulate them. The more elaborate GXL format is described in [18]. In [35] a *source code algebra* is described that can be used to express relational queries on source text. Relational algebra is used in GROK [17], Relation Manipulation Language (RML) [3], .QL [7] and Relation Partition Algebra (RPA) [11] to represent basic facts about software systems and to query them. In GUPRO [9] graphs are used to represent programs and to query them. Relations have also been proposed for software manufacture [24], software knowledge management [28], and program slicing [19]. Vankov [38] has explored the relational formulation of program slicing for different languages. His observation is also that the fact extraction phase is the major stumbling block.

In [2] set constraints are used for program analysis and type inference. More recently, we have carried out promising experiments in which the relational approach is applied to problems in software analysis [22,23] and feature analysis [37]. These experiments confirm the relevance and urgency of the research direction sketched in this paper. A formalization of fact extraction is proposed in [26].

Another approach is proposed by de Moor [6] and uses path expressions on the syntax tree to extract program facts and formulate queries on them. This approach builds on the work of Paige [34] and attempts to solve a classic problem: how to incrementally update extracted program facts (relations) after the application of a program transformation.

To conclude this brief overview, we mention one example of work that considers program analysis from the perspective of the meta-model that is used for representing extracted data. In [36] the observation is made that the meta-model needs adaptation for every analysis and proposes a method to achieve this.

1.2 Plan of the Paper

We will now first describe our approach (Section 2) and a prototype implementation (Section 3). Next we validate our approach by comparing it with other methods (Section 4) and we conclude with a discussion of our results (Section 5).

2 Description of Our Approach

In this section we will describe our fact extraction approach, called DEFacto, and show how it fits into a relational analysis process.

2.1 Requirements

Before we embark on a description of our method, we briefly summarize our requirements. The method should be:

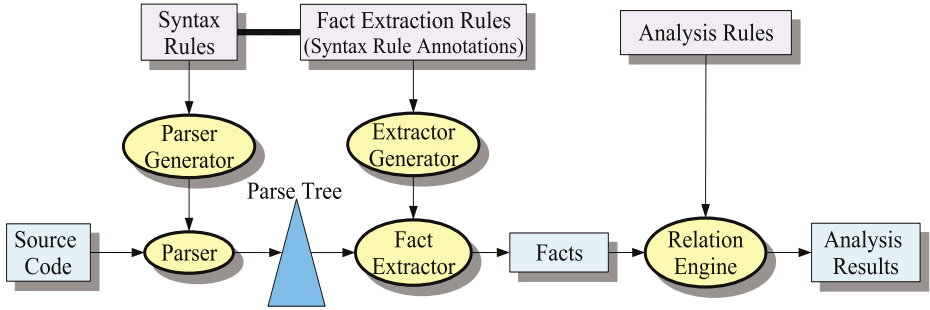


Fig. 2. Global workflow of the envisaged approach

- *language-parametric*, i.e., parametrized with the programming language(s) from which the facts are to be extracted;
- *fact-parametric*, i.e., it should be easy to extract different sets of facts for the same language;
- *local* regarding extracting facts for specific syntax rules;
- *global* when it comes to using the facts for performing analysis;
- *independent* from any specific analysis model;
- *succinct* and should have a high notional efficiency;
- completely *declarative*;
- *modular*, i.e., it should be possible to combine different sets of fact extraction rules;
- *disjoint* from the grammar so that no grammar modifications are necessary when adding fact extraction rules.

2.2 Approach

As indicated above, the main contribution of this paper is a design for a language-parametric fact extraction method. To show how it can be used to accommodate (relational) analysis, we describe the whole process from source code to analysis results. Figure 2 shows a global overview of this process.

As syntax rules we take a context free grammar of the subject system’s language. The grammar’s productions are instrumented with *fact annotations*, which declare the facts that are to be extracted from the system’s source code. We define a fact as a relation between source text elements. These elements are substrings of the text, identified by the nodes in the text’s parse tree that yield them. For instance a **declared** relation between two **Statement** nodes of the use and the declaration of a variable. With a *Relational Engine* the extracted facts are further processed and used to produce analysis results. We will discuss these steps in the following sections.

2.3 Fact Extraction with DEFACTO

Fact Annotations The fact extraction process takes as input a parse tree and a set of fact annotations to the grammar’s production rules. The annotations

declare relations between nodes of the parse tree, which identify source code elements. More precisely, a fact annotation describes relation tuples that should be created when its production rule appears in a parse tree node. This can be arbitrary n -ary tuples, consisting of the node itself, its parent, or its children. Multiple annotations can contribute tuples to the same relation.

As an example, consider the following production rule¹ for a variable declaration like, for instance, `int Counter;` or `char[100] buffer;;`

```
Type Identifier ";" -> Statement
```

A fact extraction annotation can be added to this production rule as follows:

```
Type Identifier ";" -> Statement {
    fact(typeOf, Identifier, Type)
}
```

The `fact` annotation will result in a binary relation `typeOf` between the nodes of all declared variables and their types.

In general, a fact annotation with $n + 1$ arguments declares an n -ary relation. The first argument always contains the name of the relation. The others indicate the parse tree nodes to create the relation tuples with, by referring to the production rule elements that will match these nodes. These elements are referenced using their nonterminal name, possibly followed by a number to distinguish multiple elements of the same nonterminal. List elements are postfixed by `-list` and optionals by `-opt`. The keyword `parent` refers to the parent node of the node that corresponds to the annotated production rule.

Annotation Functions. Special functions can be used to deal with the parse tree structures that lists and optionals can generate. For instance, if the above production is modified to allow the declaration of multiple variables within one statement we get:

```
Type {Identifier ","}+ ";" -> Statement {
    fact(typeOf, each(Identifier-list), Type)
}
```

Here, the use of the `each` function will extend the `typeOf` relation with a tuple for each identifier in the list. Every tuple consists of an identifier and its type. In general, each function or reference to a production rule element will yield a set (or relation). The final tuples are constructed by combining these sets using Cartesian products. Empty lists or optionals thus result in an empty set of extracted tuples.

Table 1 shows all functions that can be used in fact annotations. The functions `first`, `last` and `each` give access to list elements. The function `next` is, for instance, useful to extract the control flow of a list of statements and `index` can be useful to extract, for instance, the order of a function's parameters.

¹ Production rules are in SDF notation, so the left and right hand sides are switched when compared to BNF notation.

Table 1. Functions that can be used in fact annotations

Function	Description
<code>first()</code>	First element of a list.
<code>last()</code>	Last element of a list.
<code>each()</code>	The set of all elements of a list.
<code>next()</code>	Create a binary relation between each two succeeding elements of a list.
<code>index()</code>	Create a binary relation of type (int, node) that relates each element in a list to its index.

A function can take an arbitrary number of production rule elements as arguments. The nodes corresponding to these elements are combined into a single list before the function is evaluated. The order of the production rule elements specifies the order in which their nodes should be concatenated.

As an example, consider the Java constructor body in which the (optional) invocation of the super class constructor must be done first. This can be described by the syntax rule:

```
"{" SuperConstructorInvocation? Statement* "}" -> ConstructorBody
```

To calculate the control flow of the constructor we need the order of its contained statements. Because the `SuperConstructorInvocation` is optional and the list of regular statements can also be empty, various combinations of statements are possible. By combining all existing statements into a single list, the statement order can be extracted with only one annotation using the `next` function:

```
"{" SuperConstructorInvocation? Statement* "}" -> ConstructorBody {
    fact(succ, next(SuperConstructorInvocation-opt, Statement-list))
}
```

This results in tuples of succeeding statements to be added to the `succ` relation, only if two or more (constructor invocation) statements exist.

Selection Annotations. Sometimes however, the annotation functions might not be sufficient to extract all desired facts. This is the case when, depending on the presence or absence of nodes for a list or optional nonterminal, different facts should be extracted, but the nodes of this list or optional are not needed. In these situations the *selection annotations* `if-empty` and `if-not-empty` can be used. They take as first argument a reference to a list or optional nonterminal and as second and optionally third argument a set of annotations. If one or more parse tree nodes exist that match the first argument, the first set of annotations is evaluated, and otherwise the second set (if specified). Multiple annotations can be nested this way.

For instance, suppose the above example of a declaration statement is modified such that variables can also be declared static. If we want to extract a set (unary relation) of all static variables, this can be done as follows:

```

Static? Type Identifier ";" -> Statement {
    if-not-empty(Static-opt, [ fact(static, Identifier) ] )
}

```

Additional relations. Apart from the relations indicated with fact annotations, we also extract relations that contain additional information about each extracted node. These are binary relations that link each node to its nonterminal type, source code location (filename + coordinates) and yielded substring. Injection chains are extracted as a single node that has multiple types. This way not every injection production has to be annotated. The resulting relations also become more compact, which requires less complex analysis rules.

2.4 Decoupling Extraction Rules from Grammar Rules

Different facts are needed for different analysis purposes. Some facts are common to most analyses; use-def relations, call relations, and the control flow graph are common examples. Other facts are highly specialized and are seldomly used. For instance, calls to specific functions for memory management or locking in order to search for memory leaks or locking problems.

It is obvious that adding all possible fact extraction rules to one grammar will make it completely unreadable. We need some form of decoupling between grammar rule and fact extraction rules. It is also clear that some form of modularization is needed to enable the modular composition of fact extraction rules.

Our solution is to use an approach that is reminiscent of aspect-oriented programming. The fact extraction rules are declared separately from the grammar, in combinable modules. Grammar rules have a name² and fact extraction rules refer to the name of the grammar rule to which they are attached. Analysis rules define the facts they need and when the analysis is performed, all desired fact extraction rules are woven into the grammar and used for fact extraction. This weaving approach is well-known in the attribute grammar community and was first proposed in [8].

2.5 Relational Analysis

Fact annotations only allow the declarations of *local* relations, i.e., relations between a parse tree node and its immediate children, siblings or parent. However this is not sufficient for most fact extraction applications. For instance, the declaration and uses of a local variable can be an arbitrary number of statements apart and are typically in different branches of the parse tree.

In the analysis phase that follows fact extraction we allow the creation of relations between arbitrary parts of the programs. The extracted parse tree nodes and relations do not have to form a tree anymore. They can now be seen as (possibly disconnected) graphs, in which each node represents a source text element. Based on these extracted relations, new relations can be calculated and analyzed. Both for this enrichment of facts and for the analysis itself, we use RSCRIPT, which is explained below.

² Currently, we use the constructor attribute `cons` of SDF rules for this purpose.

The focus of fact annotations is thus local: extracting individual tuples from one syntax rule. We now shift to a more global view on the facts.

2.6 RSCRIPT at a Glance

RSCRIPT is a typed language based on relational calculus. It has some standard elementary datatypes (booleans, integers, strings) and a non-standard one: source code locations that contain a file name and text coordinates to uniquely describe a source text fragment. As composite datatypes RSCRIPT provides sets, tuples (with optionally named elements), and relations. Functions may have type parameters to make them more generic and reusable. A comprehensive set of operators and library functions is available on the built-in datatypes ranging from the standard set operations and subset generation to the manipulation of relations by taking transitive closure, inversion, domain and range restrictions and the like. The library also provide various functions (e.g., conditional reachability) that enable the manipulation of relations as graphs.

Suppose the following facts have been extracted from given source code and are represented by the relation `Calls`:

```
type proc = str
rel[proc , proc] Calls = {
  <"a", "b">, <"b", "c">, <"b", "d">,
  <"d", "c">, <"d", "e">, <"f", "e">,
  <"f", "g">, <"g", "e">}.

```

The user-defined type `proc` is an abbreviation for strings and improves both readability and modifiability of the RSCRIPT code. Each tuple represents a call between two procedures. 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. Using this knowledge, the entry points can be computed by determining the top of the `Calls` relation:

```
set[proc] entryPoints = top(Calls)

```

In this case, `entryPoints` is equal to `{"a", "f"}`. In other words, procedures "a" and "f" are the entry points of this application.

We can also determine the *indirect calls* between procedures, by taking the transitive closure of the `Calls` relation:

```
rel[proc, proc] closureCalls = Calls+

```

We know now 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 taking the *right image* of `closureCalls`. The right image operator determines all right-hand sides of tuples that have a given value as left-hand side:

```
set[proc] calledFromA = closureCalls["a"]

```

yields `{"b", "c", "d", "e"}` and

```
set[proc] calledFromF = closureCalls["f"]
```

yields {"e", "g"}. Applying this simple computation to a realistic call graph makes a good case for the expressive power and conciseness achieved in this description. In a real situation, additional information will also be included in the relation, e.g., the source code location where each procedure declaration and each call occurs.

Another feature of RSCRIPT that is relevant for this paper are the *equations*, i.e., sets of mutually recursive equations that are solved by fixed point iteration. They are typically used to define sets of dataflow equations and depend on the fact that the underlying data form a lattice.

3 A Prototype Implementation

We briefly describe a prototype implementation of our approach. With this prototype we have created two specifications for the extraction of the control flow graph (CFG) of Pico and Java programs.

3.1 Description

The prototype consists of two parts: a fact extractor and an RSCRIPT interpreter. Both are written in ASF+SDF [21,4].

DeFacto Fact Extractor. The fact extractor extracts the relevant nodes and fact relations from a given parse tree, according to a grammar and fact annotations. We currently use two tree traversals to achieve this. The first identifies all nodes that should be extracted. Each node is given a unique identifier and its non-terminal type, source location and text representation are stored. In the second traversal the actual fact relations are created. Each node with an annotated production rule is visited and its annotations are evaluated. The resulting relation tuples are stored in an intermediate relational format, called RSTORE, that is supported by the RSCRIPT interpreter. It is used to define initial values of variables in the RSCRIPT (e.g., extracted facts) and to output the values of the variables after execution of the script (e.g., analysis results). An RSTORE consists of (name, type, value) triples.

RSCRIPT interpreter. The RSCRIPT interpreter takes an RSCRIPT specification and an RSTORE as input. A typical RSCRIPT specification contains relational expressions that declare new relations, based on the contents of the relations in the given RSTORE. The interpreter calculates these declared relations, and outputs them again in RSTORE format. Since the program is written in ASF+SDF, sets and relations are internally represented as lists.

3.2 Pico Control Flow Graph Extraction

As a first experiment we have written a specification to extract the control flow graph from Pico programs. Pico is a toy language that features only three types

of statements: assignment, if-then-else and while loop. The specification consists of 13 fact annotations and only 1 RSCRIPT expression. The CFG is constructed as follows. For each statement we extract the local IN, OUT and SUCC relations. The SUCC relation links each statement to its succeeding statement(s). The IN and OUT relations link each statement to its first, respectively, last substatement. For instance, the syntax rule for the while statement is:

```
"while" Exp "do" {Statement ";" }* "od" -> Statement
```

It is annotated as follows:

```
"while" Exp "do" {Statement ";" }* "od" -> Statement {
  fact(IN, Statement, Exp),
  fact(SUCC, next(Exp, Statement-list, Exp)),
  fact(OUT, Statement, Exp)
}
```

The three extracted relations are then combined into a single graph containing only the atomic (non compound) statements, with the following RSCRIPT expression:

```
rel[node, node] basicCFG = { <N1, N4> | <node N2, node N3> : SUCC,
  node N1 : reachBottom(N2, OUT), node N4 : reachBottom(N3, IN) }
```

Where `reachBottom` is a built-in function that returns all leaf nodes of a binary relation (graph) that are reachable from a specific node. If the graph does not contain this node, the node is returned instead.

3.3 Java Control Flow Graph Extraction

After the small Pico experiment we applied our approach to a more elaborate case: the extraction of the intraprocedural control flow graph from Java programs. We wrote a DEFACTO and an RSCRIPT specification for this task, with the main purpose of comparing them (see Section 4.2) with the JastAdd specification described in [32]. We tried to resemble the output of the JastAdd extractor as close as possible.

Our specifications construct a CFG between the statements of Java methods. We first build a basic CFG containing the local order of statements, in the same way as the Pico CFG extraction described above. After that, the control flow graphs of statements with non-local behaviour (`return`, `break`, `continue`, `throw`, `catch`, `finally`) are added.

Fact annotations are used to extract information relevant for the control flow of these statements. For instance, the labels of break and continue statements, thrown expressions, and links between try, catch and finally blocks. This information is then used to modify the basic control flow graph. For each return, break, continue and throw statement we add edges that visit the statements of relevant enclosing catch and finally blocks. Then their initial successor edges are removed.

The specifications contain 68 fact annotations and 21 RSCRIPT statements, which together take up only 118 lines of code. More detailed statistics are described in section 4.

4 Experimental Validation

It is now time to compare our earlier extraction examples. In Section 4.1 we discuss an implementation in ASF+SDF of the Pico case (see Section 3.2). In Section 4.2 we discuss an implementation in JastAdd of the Java case (see Section 3.3).

4.1 Comparison with ASF+SDF

Conceptual Comparison

ASF+SDF is based on two concepts *user-definable syntax* and *conditional equations*. The user-definable syntax is provided by SDF and allows defining functions with arbitrary syntactic notation. This enables, for instance, the use of concrete syntax when defining analysis and transformation functions as opposed to defining a separate abstract syntax and accessing syntax trees via a functional interface. Conditional equations (based on ASF) provide the meaning of each function and are implemented by way of rewriting of parse trees.

Fact extraction with ASF+SDF is typically done by rewriting source code into facts, and collecting them with traversal functions. Variables have to be declared that can be used inside equations to match on source code terms. These equations typically contain patterns that resemble the production rules of the used grammar. In our approach we make use of implicit variable declaration and matching, and implicit tree traversal. We also do not need to repeat production rules, because we directly annotate them. However, ASF+SDF can match different levels of a parse tree in a single equation, which we cannot.

Pico Control Flow Extraction Using ASF+SDF

CFG extraction for Pico as described earlier in Section 3.2 can be defined in ASF+SDF by defining an extraction function `cflow` that maps language constructs to triples of type `<IN, SUCC, OUT>`. For each construct, a conditional equation has to be written that extracts facts from it and transforms these facts into a triple.

Extraction for statement sequences is done with the following conditional equation:

```
[cfg-1] <In1, Succ1, Out1> := cflow(Stat),
      <In2, Succ2, Out2> := cflow(Stats)
      =====
      cflow(Stat ; Stats) =
      < In1,
        union(Succ1, product(Out1, In2), Succ2),
        Out2 >
```

The function `cflow` is applied to the first statement `Stat`, and then to the remaining statements `Stats`. The two resulting triples are combined using relational operators to produce the triple for the complete sequence.

Extraction for while statements follows a similar pattern:

Table 2. Statistics of Pico Control Flow Graph extraction specifications

DEFACTO + RSCRIPT		ASF+SDF	
Fact extraction rules		SDF	
Fact annotations	11	Function definitions	2
Unique relations	3	Variable declarations	10
<i>Lines of code</i>	11	<i>Lines of code</i>	17
Analysis rules		ASF	
Relation expressions	1	Equations	6
<i>Lines of code</i>	2	<i>Lines of code</i>	31
Totals		Totals	
Statements	12	Statements	18
<i>Lines of code</i>	13	<i>Lines of code</i>	48

```
[cfg-3] <In, Succ, Out> := cflow(Stats),
Control := <unparse-to-string(Exp), get-location(Exp)>
=====
cflow(while Exp do Stats od) =
< {Control},
  union(product({Control}, In), Succ, product(Out, {Control})),
  {Control} >
```

The text as well as the source code location of the expression are explicitly saved in the extracted facts. Observe here (as well as in the previous equation) the use of concrete syntax in the argument of `cflow`. The text `while Exp do Stats0 od` matches a while statement and binds the variables `Exp` and `Stats0`.

Comparing the Two CFG Specifications

Using these and similar equations, leads to a simple fact extractor that can be characterized by the statistics shown in Table 2. Comparing the ASF+SDF version with our approach one can observe that the latter is shorter and that the fact extraction rules are simpler since our fact annotations have built-in functionality for building subgraphs, while this has to be spelled out in detail in the ASF+SDF version. The behaviour of our fact annotations can actually be accurately described by relational expressions as occur inside the ASF+SDF equations shown above.

The ASF+SDF version and the approach described in this paper both use SDF and do not need a specification for a separate abstract syntax.

4.2 Comparison with JastAdd

Conceptual Comparison

We have already pointed out that there is some similarity between our fact extraction rules and synthesized attributes in attribute grammars. Therefore we compare our method also with JastAdd [10], a modern attribute grammar system. The global workflow in such a system is shown in Figure 3. Given syntax rules and a definition of the desired abstract syntax tree, a parser generator

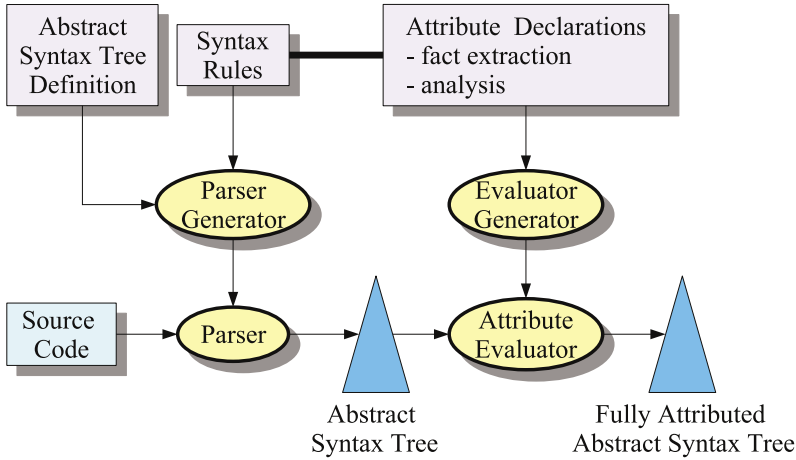


Fig. 3. Architecture of attribute-based approach

produces a parser that can transform source code into an abstract syntax tree. *Attribute Declarations* define the further processing of the tree; we focus here on fact extraction and analysis. Given the attribute definitions, an attribute evaluator is generated that repeatedly visits tree nodes until all attribute values have been computed. The primary mechanisms in any attribute grammar system are:

- *synthesized attributes*: values that are propagated from the leaves of the tree to its root.
- *inherited attributes*: values that are propagated from the root to the leaves.
- *attribute equations* define the correlation between synthesized and inherited attributes.

Due to the interplay of these mechanisms, information can be propagated between arbitrary nodes in the tree. Synthesized attributes play a dual role: for the upward propagation of facts that directly occur in the tree, and for the upward propagation of analysis results. This makes it hard to identify a boundary between pure fact extraction and the further processing of these facts. JastAdd adds to this several other mechanisms: circular attributes, collection attributes, and reference attributes, see [10] for further details.

The definitional methods used in both approaches are summarized in Table 3. The following observations can be made:

- Since we use SDF, we work on the parse tree and do not need a definition of the abstract syntax, which mostly duplicates the information in the grammar and doubles the size of the definition.
- After the extraction phase we employ a global scope on the extracted facts, so no code is needed for propagating information through an AST.

Table 3. Comparison with JastAdd

Definition	DEFACTO + RSCRIPT	JastAdd
Syntax	SDF	Any Java based parser grammar
Abstract Syntax Tree	Not needed, uses Parse Trees	AST definition + Java actions in syntax definition
Fact extraction	Modular fact extraction rules (annotation of syntax rules)	Synthesized attributes, Inherited attributes,
Analysis	RSCRIPT (relational expressions and fixed point equations)	Attribute equations, Circular attributes, Java code

- In the concept of attribute grammars the calculation of facts is scattered across different nonterminal equations, while in our approach the global scope on extracted facts allows for an arbitrary separation of concerns.
- The fixed point equations in RSCRIPT and the circular attributes in JastAdd are used for the same purpose: propagating information through the (potentially circular) control flow graph. We use the equations for reachability calculations.
- JastAdd uses Java code for AST construction as well as for attribute definitions. This gives the benefits of flexibility and tool support, but at the cost of longer specifications.
- Our approach uses less (and we, perhaps subjectively, believe simpler) definitional mechanisms, which are completely declarative. We use a grammar, fact extraction rules, and RSCRIPT while JastAdd uses a grammar, an AST definition, attribute definitions, and Java code.

Java Control Flow Extraction Using JastAdd

In [32] an implementation of intraprocedural flow analysis of Java is described, which mainly consists of CFG extraction. Here we compare its CFG extraction part to our own specification described earlier in Section 3.3.

The JastAdd CFG specification declares a **succ** attribute on statement nodes, which holds each statement's succeeding statements. Its calculation can roughly be divided into two parts: calculation of the “local” CFG and the “non-local” CFG, just like in our specification. The local CFG is stored in two helper attributes called **following** and **first**. The **following** attribute links each statement to its directly following statements. The **first** attribute contains each statement's first substatement. The following example shows the equations that define these attributes for block statements:

```
eq Block.first() = getNumStmt() > 0 ?
    SmallSet.empty().union(getStmt(0).first()) : following();
eq Block.getStmt(int i).following() = i == getNumStmt() - 1 ?
    following() : SmallSet.empty().union(getStmt(i + 1).first());
```

These attributes are similar to the (shorter) **IN** and **SUCC** annotations in our specification:

```
"{" BlockStatement* "}" -> Block {
```

```

    fact(IN, Block, first(BlockStatement-list)),
    fact(SUCC, next(BlockStatement-list)),
    fact(OUT, Block, last(BlockStatement-list))
}

```

Based on these helper attributes the **succ** attribute values are defined, which hold the entire CFG. This also includes the more elaborate control flow structures of the **return**, **break**, **continue** and **throw** statements. Due to the local nature of attribute grammars, equations can only define the **succ** attribute one edge at a time. This means that for control flow structures that pass multiple AST nodes, each node has to contribute his own outgoing edges. If multiple control flow structures pass a node, the equations on that node have to handle all these structures. For instance, the control flow of a **return** statement has to pass all **finally** blocks of enclosing **try** blocks, before exiting the function. The equations on **return** statements have to look for enclosing **try-finally** blocks, and the equations on **finally** blocks have to look for contained **return** statements. Similar constructs are required for **break**, **continue** and **throw** statements.

In our specification we calculate these non local structures at a single point in the code. For each **return** statement we construct a relation containing a path through all relevant **finally** blocks, with the following steps:

1. From a binary relation holding the scope hierarchy (consisting of blocks and **for** statements) we select the path from the root to the scope that immediately encloses the **return** statement.
2. This path is reversed, such that it leads from the **return** statement upwards.
3. From the path we extract a new path consisting only of **try** blocks that have a **finally** block.
4. We replace the **try** blocks with the internal control flow of their **finally** blocks.

The resulting relation is then added to the basic control flow graph in one go. Here we see the benefit of our global analysis approach, where we can operate on entire relations instead of only individual edges.

Comparing the Two CFG Specifications

Since both methods use different conceptual entities, it is non-trivial to make a quantitative comparison between them. Our best effort is shown in Tables 4 and 5. In Table 4, we give general metrics about the occurrence of “statements” (fact annotation, attribute equation, relational expression and the like) in both methods. Not surprisingly, the fact annotation is the dominating statement type in our approach. In JastAdd this are attribute equations. Our approach is less than half the size when measured in lines of code. The large number of lines of Java code in the JastAdd case is remarkable.

In Table 5 we classify statements per task: extraction, propagation, auxiliary statements, and calculation. For each statement type, we give a count and the percentage of the lines of code used up by that statement type. There is an interesting resemblance between our fact extraction rules and the propagation

Table 4. Statistics of Java Control Flow Graph extraction specifications

DEFACTO + RSCRIPT		JastAdd	
Fact extraction rules		Analysis rules	
Fact annotations	68	Synthesized attr. decl.	8
Selection annotations	0	Inherited attr. decl.	15
Unique relations	14	Collection attr. decl.	1
<i>Lines of code</i>	72	Unique attributes	17
Analysis rules		Equations (syn)	27
Relation expressions	19	Equations (inh)	47
Function definitions	2	Contributions	1
<i>Lines of code</i>	46	<i>Lines of Java code</i>	186
Totals		Totals	
Statements	89	Statements	99 ¹
<i>Lines of code</i>	118	<i>Lines of code</i>	287

¹ Excluding Java statements**Table 5.** Statement statistics of Java Control Flow Graph extraction specifications

	DEFACTO + RSCRIPT		JastAdd	
Extraction	68 / 61%	Fact annos 68 Selection annos 0	–	
Propagation	–		58 / 45%	Syn. attrs + eqs 14 Inh. attrs + eqs 44
Helper stats	11 / 25%	Relation exprs 10 Function defs 3	22 / 32%	Syn. attrs + eqs 4 Inh. attrs + eqs 18
Calculation	10 / 14%	Relation exprs 9 Function defs 0	19 / 23%	Syn. attrs + eqs 17 Coll. attrs + contr. 2

statements of the JastAdd specification. These propagation statements are used to “deliver” to each AST node information needed to calculate the analysis results. Interestingly, the propagated information contains no calculation results, but only facts that are immediately derivable from the AST structure. Our fact annotations also select facts from the parse tree structure, without doing any calculations. In both specifications the fact extraction and propagation take up the majority of the statements.

It is also striking that both methods need only a small fragment of their lines of code for the actual analysis 14% (Our method) versus 23% (JastAdd).

Based on these observations we conclude that both methods are largely comparable, that our method is more succinct and does not need inline Java code. We also stress that we only make a comparison of the concepts in both methods and do not yet—given the prototype state of our implementation—compare their execution efficiency.

5 Conclusions

We have presented a new technique for language-parametric fact extraction called DEFECTO. We briefly review how well our approach satisfies the requirements given in Section 2.1.

The method is certainly *language-parametric* and *fact-parametric* since it starts with a grammar and fact extraction annotations.

Fact extraction annotations are attached to a single syntax rule and result in the extraction of *local* facts from parse tree fragments. Our method does *global* relational processing of these facts to produce analysis results.

Since arbitrary fact annotations can be added to the grammar, it is *independent* from any preconceived analysis model and is fully general. The method is *succinct* and its notational efficiency has been demonstrated by comparison with other methods.

The method is *declarative* and *modular* by design and the annotations can be kept *disjoint* from the grammar in order to enable arbitrary combinations of annotations with the grammar. Observe that this solves the problem of meta-model modification in a completely different manner than proposed in [36].

The requirements we started with have indeed been met.

We have also presented a prototype implementation that is sufficient to assess the expressive power of our approach. One observation is that the intermediate RSTORE format makes it possible to completely decouple fact extraction from analysis. We have already made clear that the focus of the prototype was not on performance. Several obvious enhancements of the fact extractor can be made. A larger challenge is the efficient implementation of the relational calculator but many known techniques can be applied here. An efficient implementation is clearly one of the next things on our agenda.

Our prototype is built upon SDF, but our technique does not rely on a specific grammar formalism or parser. Also, for the processing of the extracted facts, other methods could be used as well, ranging from Prolog to Java. We intend to explore how our method can be embedded in other analysis and transformation frameworks.

The overall insight of this paper is that a clear distinction between language-parametric fact extraction and fact analysis is feasible and promising.

Acknowledgements

We appreciate discussions with Jurgen Vinju and Tijs van der Storm on the topic of relational analysis. We also thank Magiel Bruntink for his feedback on this paper. Jeroen Arnoldus kindly provided his SDFWEAVER program to us.

References

1. Aho, A.V., Kernighan, B.W., Weinberger, P.J.: Awk - a pattern scanning and processing language. *Software-Practice and Experience* 9(4), 267–279 (1979)
2. Aiken, A.: Set constraints: Results, applications, and future directions. In: Born-ing, A. (ed.) *PPCP 1994*. LNCS, vol. 874, Springer, Heidelberg (1994)

3. Beyer, D., Noack, A., Lewerentz, C.: Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering* 31(2), 137 (2005)
4. den van 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)
5. The CPPX home page (visited July 2008), <http://swag.uwaterloo.ca/~cpx/aboutCPPX.html>
6. de Moor, O., Lacey, D., van Wyk, E.: Universal regular path queries. *Higher-order and symbolic computation* 16, 15–35 (2003)
7. de Moor, O., Verbaere, M., Hajiyeve, E., Avgustinov, P., Ekman, T., Ongkingco, N., Sereni, D., Tibble, J.: Keynote address:ql for source code analysis. In: *SCAM 2007: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, Washington, DC, USA, pp. 3–16. IEEE Computer Society, Los Alamitos (2007)
8. Dueck, G.D.P., Cormack, G.V.: Modular attribute grammars. *The Computer Journal* 33(2), 164–172 (1990)
9. Ebert, J., Kullbach, B., Riediger, V., Winter, A.: GUPRO - generic understanding of programs. *Electronic Notes in Theoretical Computer Science* 72(2) (2002)
10. Ekman, T., Hedin, G.: The JastAdd system - modular extensible compiler construction. *Science of Computer Programming* 69(1–3), 14–26 (2007)
11. Feijs, L.M.G., Krikhaar, R., Ommerring, R.C.: A relational approach to support software architecture analysis. *Software Practice and Experience* 28(4), 371–400 (1998)
12. Ferenc, R., Siket, I., Gyimóthy, T.: Extracting Facts from Open Source Software. In: *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pp. 60–69. IEEE Computer Society, Los Alamitos (2004)
13. The GCC home page (visited July 2008), <http://gcc.gnu.org/>
14. GrammaTech. Codesurfer (visited July 2008), <http://www.grammatech.com/products/codesurfer/>
15. Hajiyeve, E., Verbaere, M., de Moor, O.: Codequest: Scalable source code queries with datalog. In: Thomas, D. (ed.) *Proceedings of the European Conference on Object-Oriented Programming* (2006)
16. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF - reference manual. *SIGPLAN Notices* 24(11), 43–75 (1989)
17. Holt, R.C.: Binary relational algebra applied to software architecture. CSRI 345, University of Toronto (March 1996)
18. Holt, R.C., Winter, A., Schürr, A.: GXL: Toward a standard exchange format. In: *Proceedings of the 7th Working Conference on Reverse Engineering*, pp. 162–171. IEEE Computer Society, Los Alamitos (2000)
19. Jackson, D., Rollins, E.: A new model of program dependences for reverse engineering. In: *Proc. SIGSOFT Conf. on Foundations of Software Engineering*, pp. 2–10 (1994)
20. 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)*, pp. 209–222 (1990)
21. Klint, P.: A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology* 2(2), 176–201 (1993)

22. 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)
23. Klint, P.: Using rscript for software analysis. In: *Proceedings of Query Technologies and Applications for Program Comprehension (QTAPC 2008)* (June 2008) (to appear)
24. Lamb, D.A.: *Relations in software manufacture*. Technical Report 1990-292, Queen’s University School of Computing, Kingston Ontario (1991)
25. Lesk, M.E.: *Lex - a lexical analyzer generator*. Technical Report CS TR 39, Bell Labs (1975)
26. Lin, Y., Holt, R.C.: Formalizing fact extraction. In: *ATEM 2003: First International Workshop on Meta-Models and Schemas for Reverse Engineering*, Victoria, BC, November 13 (2003)
27. Linton, M.A.: Implementing relational views of programs. In: *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pp. 132–140 (1984)
28. Meyer, B.: The software knowledge base. In: *Proceedings of the 8th international conference on Software engineering*, pp. 158–165. IEEE Computer Society Press, Los Alamitos (1985)
29. 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)
30. Murphy, G.C., Notkin, D.: Lightweight source model extraction. In: *SIGSOFT 1995: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pp. 116–127. ACM Press, New York (1995)
31. Murphy, G.C., Notkin, D., Griswold, W.G., Lan, E.S.: An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology* 7(2), 158–191 (1998)
32. Nilsson-Nyman, E., Ekman, T., Hedin, G., Magnusson, E.: Declarative intraprocedural flow analysis of java source code. In: *Proceedings of 8th Workshop on Language Descriptions, Tools and Applications (LDTA 2008)* (2008)
33. Paakki, J.: Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys* 27(2), 196–255 (1995)
34. Paige, R.: Viewing a program transformation system at work. In: *Hermenegildo, M., Penjam, J. (eds.) PLILP 1994*. LNCS, vol. 844, pp. 5–24. Springer, Heidelberg (1994)
35. Paul, S., Prakash, A.: Supporting queries on source code: A formal framework. *International Journal of Software Engineering and Knowledge Engineering* 4(3), 325–348 (1994)
36. Strein, D., Lincke, R., Lundberg, J., Löwe, W.: An extensible meta-model for program analysis. In: *ICSM 2006: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, Philadelphia, USA, pp. 380–390. IEEE Computer Society, Los Alamitos (2006)
37. van der Storm, T.: Variability and component composition. In: *Bosch, J., Krueger, C. (eds.) ICOIN 2004 and ICSR 2004*. LNCS, vol. 3107, pp. 157–166. Springer, Heidelberg (2004)
38. Vankov, I.: *Relational approach to program slicing*. Master’s thesis, University of Amsterdam (2005), <http://www.cwi.nl/~paulk/theses/Vankov.pdf>

A Case Study in Grammar Engineering

Tiago L. Alves¹ and Joost Visser²

¹ University of Minho, Portugal, and
Software Improvement Group, The Netherlands
`t.alves@sig.nl`

² Software Improvement Group, The Netherlands
`j.visser@sig.nl`

Abstract. This paper describes a case study about how well-established software engineering techniques can be applied to the development of a grammar. The employed development methodology can be described as iterative grammar engineering and includes the application of techniques such as grammar metrics, unit testing, and test coverage analysis. The result is a grammar of industrial strength, in the sense that it is well-tested, it can be used for fast parsing of high volumes of code, and it allows automatic generation of support for syntax tree representation, traversal, and interchange.

1 Introduction

Grammar engineering is an emerging field of software engineering [1] that aims to apply solid software engineering techniques to grammars, just as they are applied to other software artifacts. Such techniques include version control, static analysis, and testing. Through their adoption, the notoriously erratic and unpredictable process of developing and maintaining large grammars can become more efficient and effective, and can lead to results of higher quality. Such timely delivery of high-quality grammars is especially important in the context of grammar-centered language tool development, where grammars are used for much more than single-platform parser generation.

In this paper, we provide details of a grammar engineering project where well-established software engineering techniques were applied to the development of a VDM-SL grammar from its ISO standard language reference. Our approach can be characterised as a tool-based methodology for iterative grammar development embedded into the larger context of grammar-centered language tool development. We explain the methodology and illustrate its application in a detailed case study. We thus hope to contribute to the body of knowledge about best grammar engineering practises.

The paper is structured as follows. Section 2 describes our tool-based methodology for grammar engineering. We demonstrate key elements such as grammar versioning, metrics, visualization, and testing, and we embed our grammar engineering approach into the larger context of grammar-centered language tool development. Section 3 describes the application of these general techniques

in the specific case of our VDM grammar development project. We describe the development process and its intermediate and final deliverables. We discuss related work in Section 4, including comparison to earlier grammar engineering case studies [2,3]. In Section 5 we summarize our contributions and identify future challenges.

2 Tool-Based Grammar Engineering Methodology

Our methodology for grammar engineering is based on well-established software engineering techniques. In this section we will explain the basic ingredients of this methodology. We first explain our grammar-centered approach to language tool development and motivate our choice of grammar notation (SDF), in Section 2.1. In Section 2.2 we define the most important phases in grammar evolution which define the different kinds of transformation steps that may occur in a grammar engineering project. In Section 2.3 we present a catalogue of grammar metrics. Metrics allow quantification which is an important instrument in understanding and controlling grammar evolution. We describe tool-based grammar testing in Section 2.4 where we distinguish functional and unit testing. Finally, in Section 2.5, we discuss the use of test coverage as test quality indicator. A more general discussion of related grammar engineering work is deferred to Section 4.

2.1 Grammar-Centered Tool Development

In traditional approaches to language tool development, the grammar of the language is encoded in a parser specification. Commonly used parser generators include Yacc, Antlr, and JavaCC. The parser specifications consumed by such tools are not general context-free grammars. Rather, they are grammars within a proper subset of the class of context-free grammars, such as LL(1), or LALR. Entangled into the syntax definitions are semantic actions in a particular target programming language, such as C, C++ or Java. As a consequence, the grammar can serve only a single purpose: generate a parser in a single programming language, with a singly type of associated semantic functionality (e.g. compilation, tree building, metrics computation). For a more in-depth discussion of the disadvantages of traditional approaches to language tool development see [4].

For the development of language tool support, we advocate a *grammar-centered* approach [5]. In such an approach, the grammar of a given language takes a central role in the development of a wide variety of tools or tool components for that language. For instance, the grammar can serve as input for generating parsing components to be used in combination with several different programming languages. In addition, the grammar serves as basis for the generation of support for representation of abstract syntax, serialization and de-serialization in various formats, customizable pretty-printers, and support for syntax tree traversal. This approach is illustrated by the diagram in Figure 1.

For the description of grammars that play such central roles, it is essential to employ a grammar description language that meets certain criteria. It must be neutral with respect to target implementation language, it must not impose

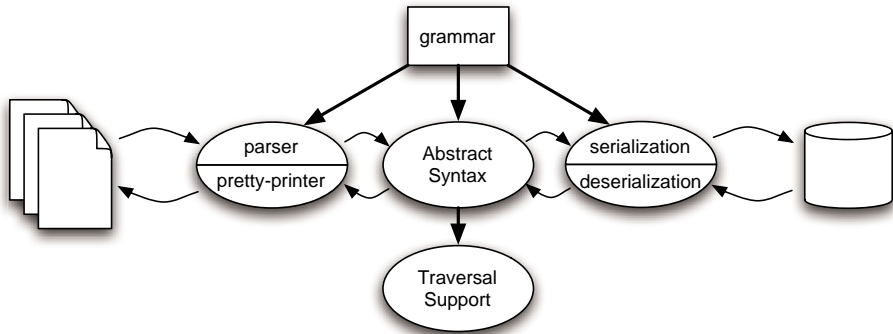


Fig. 1. Grammar-centered approach to language tool development

restrictions on the set of context-free languages that can be described, and it should allow specification not of semantics, but of syntax only. Possible candidates are BNF or EBNF, or our grammar description of choice: SDF [6,7].

The syntax definition formalism SDF allows description of both lexical and context-free syntax. It adds even more regular expression-style construct to BNF than EBNF does, such as separated lists. It offers a flexible modularization mechanism that allows modules to be mutually dependent, and distribution of alternatives of the same non-terminal across multiple modules. Various kinds of tool support are available for SDF, such as a well-formedness checker, a GLR parser generator, generators of abstract syntax support for various programming languages, among which Java, Haskell, and Stratego, and customizable pretty-printer generators [8,9,10,11,12].

2.2 Grammar Evolution

Grammars for sizeable languages are not created instantaneously, but through a prolonged, resource consuming process. After an initial version of a grammar has been created, it goes through an evolutionary process, where piece-meal modifications are made at each step. After delivery of the grammar, evolution may continue in the form of corrective and adaptive maintenance.

A basic instrument in making such evolutionary processes tractable is version control. We have chosen the Concurrent Versions System (CVS) as the tool to support such version control [13].

As alternative, GDK [14] could be used enabling the documentation of changes as a set of transformations. Although this is an interesting academic approach we preferred the use of more practical and standard tool.

In grammar evolution, different kinds of transformation steps occur:

Recovery: An initial version of the grammar may be retrieved by reverse engineering an existing parser, or by converting a language reference manual, available as Word or PDF document. If only a hardcopy is available then it should be transcribed.

Size and complexity metrics	
TERM	Number of terminals
VAR	Number of non-terminals
MCC	McCabe's cyclomatic complexity
AVS-P	Average size of RHS per production
AVS-N	Average size of RHS per non-terminal

Fig. 2. Size and complexity metrics for grammars

Error correction: Making the grammar complete, fully connected, and correct by supplying missing production rules, or adapting existing ones.

Extension or restriction: Adding rules to cover the constructs of an extended language, or removing rules to limit the grammar to some core language.

Refactoring: Changing the shape of the grammar, without changing the language that it generates. Such shape changes may be motivated by different reasons. For instance, changing the shape may make the description more concise, easier to understand, or it may enable subsequent correction, extensions, or restrictions.

In our case, grammar descriptions will include disambiguation information, so adding disambiguation information is yet another kind of transformation step present in our evolution process.

2.3 Grammar Metrics

Quantification is an important instrument in understanding and controlling grammar evolution, just as it is for software evolution in general. We have adopted, adapted, and extended the suite of metrics defined for BNF in [15] and implemented a tool, called SdfMetz, to collect grammar metrics for SDF.

Adaptation was necessary because SDF differs from (E)BNF in more than syntax. For instance, it allows several productions for the same non-terminal. This forced us to choose between using the number of productions or the number of non-terminals in some metrics definitions.

Furthermore, SDF grammars contain more than just context-free syntax. They also contain lexical syntax and disambiguation information. We decided to apply the metrics originally defined for BNF only to the context-free syntax, to make comparisons possible with the results of others. For the disambiguation information these metrics were extended with the definition of a dedicated set of metrics. Full details about the definition and the implementation of these SDF metrics are provided in [16].

We will discuss several categories of metrics: size and complexity metrics, structure metrics, Halstead metrics, and disambiguation metrics by providing a brief description of each.

Size, Complexity, and Structure Metrics. Figure 2 lists a number of size and complexity metrics for grammars. These metrics are defined for BNF in [15]. The number of terminals (TERM) and non-terminals (VAR) are simple metrics

Structure metrics	
TIMP	Tree impurity (%)
CLEV	Normalized count of levels (%)
NSLEV	Number of non-singleton levels
DEP	Size of largest level
HEI	Maximum height

Fig. 3. Structure metrics for grammars

applicable equally to BNF and SDF grammars. McCabe’s cyclomatic complexity (MCC), originally defined for program complexity, was adapted for BNF grammars, based on an analogy between grammar production rules and program procedures. Using the same analogy, MCC can be extended easily to cover the operators that SDF adds to BNF.

The average size of right-hand sides (AVS) needs to be adapted to SDF with more care. In (E)BNF the definition of AVS is trivial: count the number of terminals and non-terminals on the right-hand side of each grammar rule, sum these numbers, and divide them by the number of rules. In SDF, this definition can be interpreted in two ways, because each non-terminal can have several productions associated to it. Therefore, we decided to split AVS into two separate metrics: average size of right-hand sides per production (AVS-P) and average size of right-hand sides per non-terminal (AVS-N). For grammars where each non-terminal has a single production rule, as is the case for (E)BNF grammars, these metrics will present the same value. For SDF grammars, the values can be different. While the AVS-N metric is more appropriate to compare with other formalisms (like BNF and EBNF), the AVS-P metric provides more precision.

Structure Metrics. Figure 3 lists a number of structure metrics also previously defined in [15]. Each of these metrics is based on the representation of a grammar as a graph which has non-terminals as nodes, and which contains edges between two non-terminals whenever one occurs in the right-hand side of the definition of the other. This graph is called the grammar’s flow graph.

Only the tree impurity metric (TIMP) is calculated directly from this flow graph, all the other structure metrics are calculated from the corresponding strongly connected components graph. This latter graph is obtained from the flow graph by grouping the non-terminals that are strongly connected (reachable from each other) into nodes (called components of levels of the grammar). An edge is created from one component to another if in the flow graph at least one non-terminal from one component has an edge to a non-terminal from the other component. This graphs is called the grammar’s level graph or graph of strongly connected components.

Tree impurity (TIMP) measures how much the flow graph deviates from a tree, expressed as a percentage. A tree impurity of 0% means that the graph is a tree and a tree impurity of 100% means that is a fully connected graph.

Normalized count of levels (CLEV) expresses the number of nodes in the level graph (graph of strongly connected components) as a percentage of the number

Halstead basic metrics	
n1	Number of distinct operators
n2	Number of distinct operands
N1	Total number of operators
N2	Total number of operands

Halstead derived metrics	
n	Program vocabulary
N	Program length
V	Program volume
D	Program difficulty
E	Program effort (HAL)
L	Program level
T	Program time

Fig. 4. Halstead metrics

of nodes in the flow graph. A normalized count of levels of 100% means that there are as many levels in the level graph as non-terminals in the flow graph. In other words, there are no circular connections in the flow graph, and the level graph only contains singleton components. A normalized count of levels of 50% means that about half of the non-terminals of the flow graph are involved in circularities and are grouped into non-singleton components in the level graph.

Number of non-singleton levels (NSLEV) indicates how many of the grammar strongly connected components (levels) contain more than a single non-terminal.

Size of the largest level (DEP) measures the depth (or width) of the level graph as the maximum number of non-terminals per level.

Maximum height (HEI) measures the height of the level graph as the longest vertical path through the level graph, i.e. the biggest path length from a source of the level graph to a sink.

Halstead Metrics. The Halstead Effort metric [17] has also been adapted for (E)BNF grammars in [15]. We compute values not only for Halstead’s effort metric but also for some of its ingredient metrics and related metrics. Figure 4 shows a full list. The essential step in adapting Halstead’s metrics to grammars is to interpret the notions of *operand* and *operator* in the context of grammars. For more details about the interpretation from BNF to SDF we refer again to [16].

The theory of software science behind Halstead’s metrics has been widely questioned. In particular, the meaningfulness and validity of the effort and time metrics have been called into question [18]. Below, we will still report HAL, for purposes of comparison to data reported in [15].

Disambiguation Metrics. In SDF, disambiguation constructs are provided in the same formalism as the syntax description itself. To quantify this part of SDF grammars, we defined a series of metrics, which are shown in Figure 5. These metrics are simple counters for each type of disambiguation construct offered by the SDF notation.

2.4 Grammar Testing

In software testing, a global distinction can be made between white box testing and black box testing. In black box testing, also called functional or behavioral testing, only the external interface of the subject system is available. In white box

Disambiguation metrics	
FRST	Number of follow restrictions
ASSOC	Number of associativity attributes
REJP	Number of reject productions
UPP	Number of unique productions in priorities

Fig. 5. Disambiguation metrics for grammars

testing, also called unit testing, the internal composition of the subject system is taken into consideration, and the individual units of this composition can be tested separately.

In grammar testing, we make a similar distinction between functional tests and unit tests. A functional grammar test will use complete files as test data. The grammar is tested by generating a parser from it and running this parser on such files. Test observations are the success or failure of the parser on a input file, and perhaps its time and space consumption. A unit test will use fragments of files as test data. Typically, such fragments are composed by the grammar developer to help him detect and solve specific errors in the grammar, and to protect himself from reintroducing the error in subsequent development iterations. In addition to success and failure observations, unit tests may observe the number of ambiguities that occur during parsing, or the shape of the parse trees that are produced.

For both functional and unit testing we have used the **parse-unit** utility [19]. Tests are specified in a simple unit test description language with which it is possible to declare whether a certain input should parse or not, or that a certain input sentence should produce a specific parse tree (tree shape testing). Taking such test descriptions as input, the **parse-unit** utility allows batches of unit tests to be run automatically and repeatedly.

2.5 Coverage Metrics

To determine how well a given grammar has been tested, a commonly used indicator is the number of non-empty lines in the test suites.

A more reliable instrument to determine grammar test quality is coverage analysis. We have adopted the rule coverage (RC) metric [20] for this purpose. The RC metric simply counts the number of production rules used during parsing of a test suite, and expresses it as a percentage of the total number of production rules of the grammar.

SDF allows two possible interpretations of RC, due to the fact that a single non-terminal may be defined by multiple productions. (Above, we discussed a similar interpretation problem for the AVS metric.) One possibility is to count each of these alternative productions separately. Another possibility is to count different productions of the same non-terminal as one. For comparison with rule coverage for (E)BNF grammars, the latter is more appropriate. However, the former gives a more accurate indication of how extensively a grammar is covered by the given test suite. Below we report both, under the names of RC (rule

coverage) and NC (non-terminal coverage), respectively. These numbers were computed for our functional test suite and unit test suite by a tool developed for this purpose, called SdfCoverage [16].

An even more accurate indication can be obtained with context-dependent rule coverage [21]. This metric takes into account not just whether a given production is used, but also whether it has been used in each context (use site) where it can actually occur. However, implementation and computation of this metric is more involved.

3 Development of the VDM Grammar

The Vienna Development Method (VDM) is a collection of techniques for the formal specification and development of computing systems. VDM consists of a specification language, called VDM-SL, and an associated proof theory. Specifications written in VDM-SL describe mathematical models in terms of data structures and data operations that may change the model's state. VDM-SL is quite a rich language in the sense that its syntax provides notation for a wide range of basic types, type constructors, and mathematical operators (the size and complexity of the language will be quantified below). VDM's origins lie in the research on formal semantics of programming languages at IBM's Vienna Laboratory in the 1960s and 70s, in particular of the semantics of PL/I. In 1996, VDM achieved ISO standardization.

We have applied the grammar engineering techniques described above during the iterative development of an SDF grammar of VDM-SL. This grammar development project is ancillary to a larger effort in which various kinds of VDM tool support are developed in the context of Formal Methods research. For example, the grammar has already been used for the development of VooDOOM, a tool for converting VDM-SL specifications into relational models in SQL [22].

In this section we describe the scope, priorities, and planned deliverables of the project, as well as its execution. We describe the evolution of the grammar during its development both in qualitative and quantitative terms, using the metrics described above. The test effort during the project is described in terms of the test suites used and the evolution of the unit tests and test coverage metrics during development.

3.1 Scope, Priorities, and Planned Deliverables

Language. Though we are interested in eventually developing grammars for various existing VDM dialects, such as IFAD VDM and VDM++, we limited the scope of the initial project to the VDM-SL language as described in the ISO VDM-SL standard [23].

Grammar shape. Not only should the parser generate the VDM-SL language as defined in the standard, we also want the shape of the grammar, the names of the non-terminals, and the module structure to correspond closely to the grammar. We want to take advantage of SDF's advanced regular expression-style constructs wherever this leads to additional conciseness and understandability.

Parsing and parse trees. Though the grammar should be suitable for generation of a wide range of tool components and tools, we limited the scope of the initial project to development of a grammar from which a GLR parser can be generated. The generated parser should be well-tested, exhibit acceptable time and space consumption, parse without ambiguities, and build abstract syntax trees that correspond as close as possible to the abstract syntax as defined in the standard.

Planned deliverables. Based on the defined scope and priorities, a release plan was drawn up with three releases within the scope of the initial project:

Initial Grammar. Straightforward transcription of the concrete syntax BNF specification of the ISO standard into SDF notation. Introduction of SDF's regular expression-style constructs.

Disambiguated Grammar. Addition of disambiguation information to the grammar, to obtain a grammar from which a non-ambiguous GLR parser can be generated.

Refactored Grammar. Addition of constructor attributes to context-free productions to allow generated parsers to automatically build ASTs with constructor names corresponding to abstract syntax of the standard. Changes in the grammar shape to better reflect the tree shape as intended by the abstract syntax in the standard.

The following functionalities have explicitly been kept outside the scope of the initial project, and are planned to be added in follow-up projects:

- Haskell front-end¹. Including generated support for parsing, pretty-printing, AST representation, AST traversal, marshalling ASTs to XML format and back, marshalling of ASTs to ATerm format and back.
- Java front-end. Including similar generated support.
- IFAD VDM-SL extensions, including module syntax.
- VDM object-orientation extension (VDM⁺⁺).

Section 5.2 discusses some of this future work in more detail.

3.2 Grammar Creation and Evolution

To accurately keep track of all grammar changes, a new revision was created for each grammar evolution step. This led to the creation of a total of 48 development versions. While the first and the latest release versions (initial and refactored) correspond to development versions 1 and 48 of the grammar, respectively, the intermediate release version (disambiguated) corresponds to version 32.

The Initial Grammar. The grammar was transcribed from the hardcopy of the ISO Standard [23]. In that document, context-free syntax, lexical syntax and disambiguation information are specified in a semi-formal notation. Context-free syntax is specified in EBNF, but the terminals are specified as mathematical

¹ At the time of writing, a release with this functionality has been completed.

symbols. To translate the mathematical symbols to ASCII symbols an interchange table is provided. Lexical syntax is specified in tables by enumerating the possible symbols. Finally, disambiguation information is specified in terms of precedence in tables and equations.

Apart from changing syntax from EBNF to SDF and using the interchange table to substitute mathematical symbols for its parseable representation, the following was involved in the transcription.

Added SDF Constructs. Although a direct transcription from the EBNF specification was possible, we preferred to use SDF specific regular-expression-style constructs. For instance consider the following excerpt from the ISO VDM-SL EBNF grammar:

```
product type = type, "*", type, { "*", type} ;
```

During transcription this was converted to:

```
{ Type "*" }2+ -> ProductType
```

Both excerpts define the same language. Apart from the syntactic differences from EBNF to SDF, the difference is that SDF has special constructs that allows definition of the repetition of a non-terminal separated by a terminal. In this case, the non-terminal **Type** appears at least two times and is always separated by the terminal **"*"**.

Detected Top and Bottom Non-terminals. To help the manual process of typing the grammar a small tool was developed to detect top and bottom non-terminals. This tool helped to detect typos. More than one top non-terminal, or a bottom non-terminal indicates that a part of the grammar is not connected. This tool provided a valuable help not only in this phase but also during the overall development of the grammar.

Modularized the Grammar. (E)BNF does not support modularization. The ISO Standard separates concerns by dividing the EBNF rules over sections. SDF does support modules, which allowed us to modularize the grammar following the sectioning of the ISO standard. Also, another small tool was implemented to discover the dependencies between modules.

Added Lexical Syntax. In SDF, lexical syntax can be defined in the same grammar as context-free syntax, using the same notation. In the ISO standard, lexical syntax is described in an ad hoc notation resembling BNF, without clear semantics. We interpreted this lexical syntax description and converted it into SDF. Obtaining a complete and correct definition required renaming some lexical non-terminals and providing additional definitions. Detection of top and bottom non-terminals in this case helped to detect some inconsistencies in the standard.

Disambiguation. In SDF, disambiguation is specified by means of dedicated disambiguation constructs [24]. These are specified more or less independently from the context-free grammar rules. The constructs are associativity attributes, priorities, reject productions and lookahead restrictions.

Table 1. Grammar metrics for the three release versions

Version	TERM	VAR	MCC	AVS-N	AVS-P	HAL	TIMP	CLEV	NSLEV	DEP	HEI
initial	138	161	234	4.4	2.3	55.4	1%	34.9	4	69	16
disambiguated	138	118	232	6.4	2.8	61.1	1.5%	43.9	4	39	16
refactored	138	71	232	10.4	3.3	68.2	3%	52.6	3	27	14

In the ISO standard, disambiguation is described in detail by means of tables and a semi-formal textual notation. We interpreted these descriptions and expressed them with SDF disambiguation constructs. This was not a completely straightforward process, in the sense that it is not possible to simply translate the information of the standard document to SDF notation. In some cases, the grammar must respect specific patterns in order enable disambiguation. For each disambiguation specified, a unit test was created.

Refactoring. As already mentioned, the purpose of this release was to automatically generate ASTs that follow the ISO standard as close as possible. Two operations were performed. First, constructor attributes were added to the context-free rules to specify AST node labels. Second, injections were removed to make the grammar and the AST's nicer.

The removal of the injections needs further explanation. We call a production rule an injection when it is the only defining production of its non-terminal, and its right-hand side contains exactly one (different) non-terminal. Such injections, which already existed in the original EBNF grammar, were actively removed, because they needlessly increase the size of the grammar (which can be observed in the measurements) and reduce its readability. Also, the corresponding automatically built ASTs are more compact after injection removal. The names of the inlined productions, however, were preserved in annotations to create the AST, so actual no information loss occurs.

3.3 Grammar Metrics

We measured grammar evolution in terms of the size, complexity, structure, and Halstead metrics introduced above. The data is summarized in Table 1. This table shows the values of all metrics for the three released versions. In addition, Figure 6 graphically plots the evolution of a selection of the metrics for all 48 development versions.

Size and Complexity Metrics. A first important observation to make is that the number of terminals (TERM) is constant throughout grammar development. This is conform to expectation, since all keywords and symbols of the language are present from the first grammar version onward.

The initial number of 161 non-terminals (VAR) decreases via 118 after disambiguation to 71 after refactoring. These numbers are the consequence of changes in grammar shape where non-terminals are replaced by their definition. In the disambiguation phase (43 non-terminals removed), such non-terminal inlining

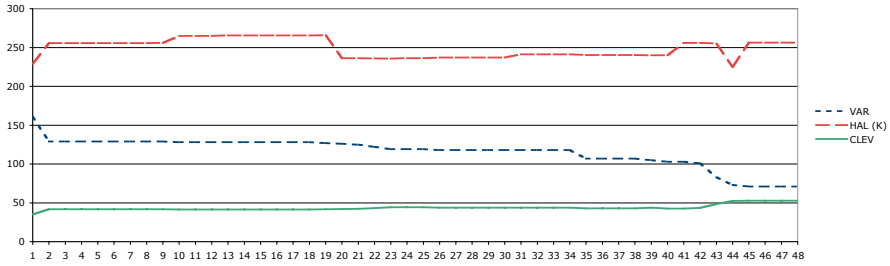


Fig. 6. The evolution of VAR, HAL, and CLEV grammar metrics during development. The x-axis represents the 48 development versions.

was performed to make formulation of the disambiguation information possible, or easier. For instance, after inlining, simple associativity attributes would suffice to specify disambiguation, while without inlining more elaborate reject productions might have been necessary. In the refactoring phase (47 non-terminals removed), the inlinings performed were mainly removals of injections. These were performed to make the grammar easier to read, more concise, and suitable for creation of ASTs closer to the abstract syntax specification in the standard.

The value of the McCabe cyclomatic complexity (MCC) is expected to remain constant. However the value decreases by 2 during disambiguation, meaning that we eliminated two paths in the flow graph of the grammar. This was caused by refactoring the syntax of product types and union types in similar ways required for disambiguation purposes. In case of product types, the following two production rules:

```
ProductType    -> Type
{ Type "*" }2+ -> ProductType
```

were replaced by a single one:

```
Type "*" Type -> Type
```

For union types, the same replacement was performed. The language generated by the grammar remained the same after refactoring, but disambiguation using priorities became possible. In refactorings, MCC remains constant as expected.

The average rules size metrics, AVS-N and AVS-P increase significantly. These increases are also due to inlining of non-terminals. Naturally, when a non-terminal with a right-hand size of more than 1 is inlined, the number of non-terminals decreases by 1, and the size of the right-hand sides of the productions in which the non-terminal was used goes up. The increase of AVS-N is roughly by a factor of 2.4, while the increase of AVS-P is by a factor of 1.4. Given the decrease of VAR, increase in AVS-N is expected. However, the limited increase in AVS-P indicates that inlining did not produce overly large production rules.

Halstead Metrics. The value of the Halstead Effort metric (HAL) fluctuates during development. It starts at 228K in the initial grammar, and immediately

risers to 255K. This initial rise is directly related to the removal of 32 non-terminals. The value then rises more calmly to 265K, but drops again abruptly towards the end of the disambiguation phase, to the level of 236K. During refactoring, the value rises again to 255K, drops briefly to 224K, and finally stabilizes at 256K. Below, a comparison of these values with those of other grammars will be offered. The use of Halstead is for comparison purpose only and we attach no conclusions.

Structure Metrics. Tree impurity (TIMP) measures how much the grammar's flow graph deviates from a tree, expressed as a percentage. The low values for this measure indicates that our grammar is almost a tree, or, in other words, that complexity due to circularities is low. As the grammar evolves, the tree impurity increases steadily, from little more than 1%, to little over 3%. This development can be attributed directly to the non-terminal inlining that was performed. When a non-terminal is inlined, the flow graph becomes smaller, but the number of cycles remains equal, i.e. the ratio of the latter becomes higher.

Normalized count of levels (CLEV) indicates roughly the percentage of modularizability, if grammar levels (strongly connected components in the flow graph) are considered as modules. Throughout development, the number of levels goes down (from 58 to 40; values are not shown), but the *potential* number of levels, i.e. the number of non-terminals, goes down more drastically (from 161 to 71). As a result, CLEV rises from 34% to 53%, meaning that the percentage of modularizability increases.

The number of non-singleton levels (NSLEV) of the grammar is 4 throughout most of its development, except at the end, where it goes down to 3. Inspection of the grammar learns us that these 4 levels roughly correspond to *Expressions*, *Statement*, *Type* and *StateDesignators*. The latter becomes a singleton level towards the end of development due to inlining.

The size of the largest grammar level (DEP) starts initially very high at 69 non-terminals, but drops immediately to only 39. Towards the end of development, this number drops further to 27 non-terminals in the largest level, which corresponds to *Expressions*. The decrease in level sizes is directly attributable to inlining of grammar rules involved in cycles.

The height of the level graph (HEI) is 16 throughout most of the evolution of the grammar, but sinks slightly to 14 towards the end of development. Only inlining of production rules not involved in cycles leads to reduction of path length through the level graph. This explains why the decrease of HEI is modest.

Disambiguation Metrics. In Figure 7 we plot the evolution of two disambiguation metrics and compare them to the number of productions metric (PROD). Although we computed more metrics, we chose to show only the number of associativity attributes (ASSOC) and the number of unique productions in priorities (UPP) because these are the two types of disambiguation information most used during the development.

In the disambiguation phase, 31 development versions were produced (version 32 corresponds to the disambiguated grammar). Yet, by analyzing the chart we

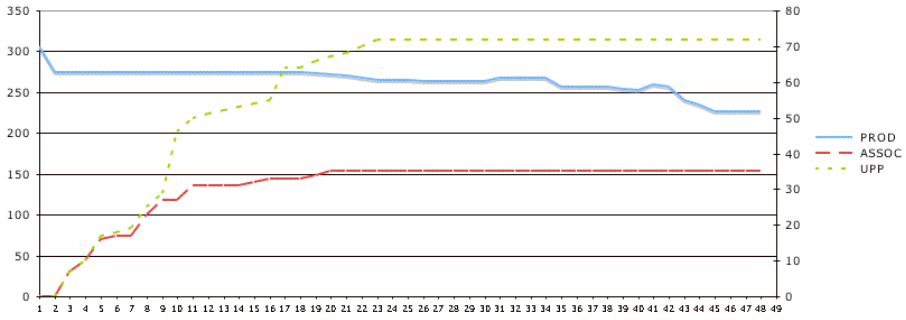


Fig. 7. The evolution of the ASSOC and UPP disambiguation metrics compared with the evolution of the number of productions (PROD). The x-axis represents the 48 development versions.

can see that the ASSOC and the UPP metrics stabilize after the 23rd version. This is because after this version other kind of disambiguation information was added (reject productions and lookahead restrictions) which are not covered by the chosen metrics.

Also, it is interesting to see that in the development version 2 no disambiguation information was added but the number of productions drops significantly. This was due to injection removal necessary to prepare for disambiguation.

From the version 2 to 9, the UPP and ASSOC metrics grow, in most cases, at the same rate. In these steps, the binary expressions were disambiguated using associativity attributes (to remove ambiguity between a binary operator and itself) and priorities (to remove ambiguity between a binary operator and other binary operators). Between version 9 to 10, a large number of unary expressions were disambiguated, involving priorities (between the unary operator and binary operators) but not associativity attributes (unary operators are not ambiguous with themselves).

From versions 9 to 16 both metrics increase fairly gradually. But in version 17, there is a surge in the number of productions in priorities. This was caused by the simultaneous disambiguation of a group of expressions with somewhat similar syntax (let, def, if, foreach, exits, etc. expressions) which do not have associativity information.

From versions 18 to 24 the number of production in priorities grows simultaneously while the total number of productions decreases. This shows that, once more, disambiguation was only enabled by injection removal or by inlining.

Although not shown in the chart, from the 24th version until the 32th disambiguation was continued by adding lookahead restrictions and reject productions. In this phase lexicals were disambiguated by keyword reservation or by preferring the longer match in lexical recognition. For that reason, the total number of productions remains practically unchanged.

Table 2. Grammar metrics for VDM and other grammars. The italicized grammars are in BNF, and their metrics are reproduced from [15]. The remaining grammars are in SDF. Rows have been sorted by Halstead effort (HAL), reported in thousands.

Grammar	TERM	VAR	MCC	AVS-N	AVS-P	HAL	TIMP	CLEV	NSLEV	DEP	HEI
Fortran 77	21	16	32	8.8	3.4	26	11.7	95.0	1	2	7
<i>ISO C</i>	86	65	149	5.9	5.9	51	64.1	33.8	3	38	13
<i>Java v1.1</i>	100	149	213	4.1	4.1	95	32.7	59.7	4	33	23
AT&T SDL	83	91	170	5.0	2.6	138	1.7	84.8	2	13	15
<i>ISO C⁺⁺</i>	116	141	368	6.1	6.1	173	85.8	14.9	1	121	4
<i>ECMA Standard C[#]</i>	138	145	466	4.7	4.7	228	29.7	64.9	5	44	28
ISO VDM-SL	138	71	232	10.4	3.3	256	3.0	52.6	3	27	14
VS Cobol II	333	493	739	3.2	1.9	306	0.24	94.4	3	20	27
VS Cobol II (<i>alt</i>)	364	185	1158	10.4	8.3	678	1.18	82.6	5	21	15
PL/SQL	440	499	888	4.5	2.1	715	0.3	87.4	2	38	29

Grammar Comparisons. In this section we compare our grammar, in terms of metrics, to those developed by others in SDF, and in Yacc-style BNF. The relevant numbers are listed in Table 2, sorted by the value of the Halstead Effort.

The numbers for the grammars of C, Java, C⁺⁺, and C[#] are reproduced from the same paper from which we adopted the various grammar metrics [15]. These grammars were specified in BNF, or Yacc-like BNF dialects. Note that for these grammars, the AVS-N and AVS-P metrics are always equal, since the number of productions and non-terminals is always equal in BNF grammars.

The numbers for the remaining grammars were computed by us. These grammars were all specified in SDF by various authors. Two versions of the VS Cobol II grammar are listed: *alt* makes heavy use of nested alternatives, while in the other one, such nested alternatives have been folded into new non-terminals.

Note that the tree impurity (TIMP) values for the SDF grammars are much smaller (between 0.2% and 12%) than for the BNF grammars (between 29% and 86%). This can be attributed to SDF's extended set of regular expression-style constructs, which allow more kinds of iterations to be specified without (mutually) recursive production rules. In general, manually coded iterations are error prone and harder to understand. This can be revealed by high values of the TIMP metric.

In terms of Halstead effort, our VDM-SL grammar ranks quite high, only behind the grammars of the giant Cobol and PL/SQL languages.

Discussion. As previously observed, different metrics can play different roles, namely validation, quality improvement, productivity and comparison. Examples of validation are MCC and TERM which are expected to remain constant under semantics preserving operations such as disambiguations and refactorings. If deviations are noticed, inspection is needed to check no errors were committed. Examples of quality improvement are VAR, DEP and HEI. Lower values mean a more compact and easier to understand grammar. The increase of TIMP can be seen as quality loss, but is explainable against background of strongly decreasing

Table 3. Integration test suite. The second column gives the number of code lines. The third and fourth columns gives coverage values for the final grammar.

Origin	LOC	RC	NC
Specification of the MAA standard (Graeme Parkin)	269	19%	30%
Abstract data types (Matthew Suderman and Rick Sutcliffe)	1287	37%	53%
A crosswords assistant (Yves Ledru)	144	28%	43%
Modelling of Realms in (Peter Gorm Larsen)	380	26%	38%
Exercises formal methods course Univ. do Minho (Tiago Alves)	500	35%	48%
Total	2580	50%	70%

VAR, so no actual quality is lost. Examples of productivity are disambiguation metrics such as ASSOC and UPP. Finally, for comparison in order to be able to quantitatively and qualitatively rank different grammars.

3.4 Test Suites

Integration Test Suite. The body of VDM-SL code that strictly adheres to the ISO standard is rather small. Most industrial applications have been developed with tools that support some superset or other deviation from the standard, such as VDM⁺⁺. We have constructed an integration test suite by collecting specifications from the internet². A preprocessing step was done to extract VDM-SL specification code from literate specifications. We manually adapted specifications that did not adhere to the ISO standard.

Table 3 lists the suite of integration tests that we obtained in this way. The table also shows the lines of code (excluding blank lines and comments) that each test specification contains, as well as the rule coverage (RC) and non-terminal coverage (NC) metrics for each. The coverage metrics shown were obtained with the final, refactored grammar.

Note that in spite of the small size of the integration test suite in terms of lines of code, the test coverage it offers for the grammar is satisfactory. Still, since test coverage is not 100%, a follow-up project specifically aimed at enlarging the integration test suite would be justified.

Unit Tests. During development, unit tests were created incrementally. For every problem encountered, one or more unit tests were created to isolate the problem.

We measured unit tests development during grammar evolution in terms of lines of unit test code, and coverage by unit tests in terms of rules (RC) and non-terminals (NC). This development is shown graphically in Figure 8. As the chart indicates, all unit tests were developed during the disambiguation phase, i.e. between development versions 1 and 32. There is a small fluctuation in the beginning of the disambiguation process that is due to unit-test strategy changes. Also, between version 23 and 32, when lexical disambiguation was carried out, unit tests were not added due to limitations of the test utilities.

² A collection of specifications is available from <http://www.csr.ncl.ac.uk/vdm/>

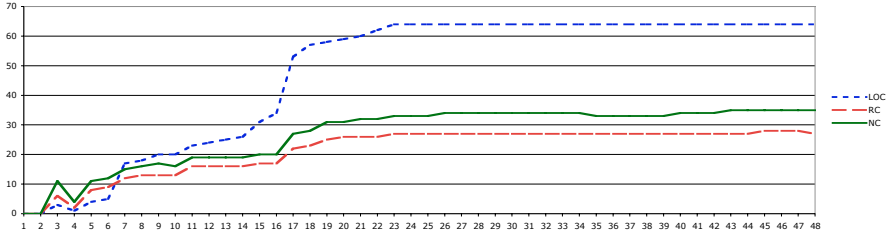


Fig. 8. The evolution of unit tests during development. The x-axis represents the 48 development versions. The three release versions among these are 1, 32, and 48. The left y-axis corresponds to lines of unit test code. Rule coverage (RC) and non-terminal coverage (NC) are shown as well.

During the refactoring phase, the previously developed unit tests were used to prevent introducing errors unwittingly. Small fluctuations of coverage metrics during this phase are strictly due to variations in the total numbers of production rules and non-terminals.

4 Related Work

Malloy and Power have applied various software engineering techniques during the development of a LALR parser for C# [2]. Their techniques include versioning, testing, and the grammar size, complexity, and some of the structure metrics that we adopted ([15], see Table 2). For Malloy and Power, an important measure of grammar quality is the number of conflicts. In our setting, ambiguities play a prominent role, rather than conflicts, due to the use of a *generalized* LR parser generation technology. We use test coverage as an additional measure of grammar quality. Also, we develop unit tests and we use them in addition to integration tests. Finally, we monitored a richer set of metrics during grammar evolution and we reported how to use them to validate grammar changes.

Lämmel et al. have advocated derivation of grammars from language reference documents through a semi-automatic transformational process [3,25]. In particular, they have applied their techniques to recover the VS COBOL II grammar from railroad diagrams in an IBM manual. They use metrics on grammars, though less extensive than we. Coverage measurement nor tests are reported.

Klint et al. provide a survey over grammar engineering techniques and a agenda for grammar engineering research [1]. In the area of natural language processing, the need for an engineering approach and tool support for grammar development has also been recognized [26,27].

5 Concluding Remarks

With the detailed grammar engineering case study presented in this paper, we have contributed to the grammar engineering body of knowledge in several ways.

5.1 Contributions

We showed how grammar testing, grammar metrics, and coverage analysis can be combined in systematic grammar development process that can be characterised as a tool-based methodology for iterative grammar development. We have motivated the use of these grammar engineering techniques from the larger context of grammar-centered language tool development, and we have conveyed our experiences in using them in a specific grammar development project. We have demonstrated that this approach can help to make grammar development a controlled process, rather than a resource-intensive and high-risk adventure.

The approach that we explained and illustrated combines several techniques, such as version control, unit testing, metrics, and coverage analysis. We have demonstrated how the development process can be monitored with various grammar metrics, and how tests can be used to guide the process of grammar disambiguation and refactorization. The presented metrics values quantify the size and complexity of the grammar and reveal the level of continuity during evolution.

We have indicated how this case study extends earlier case studies [2,3] where some of these techniques were demonstrated before. Novelties in our approach include the extensive use of unit tests, combined with monitoring of several test coverage indicators. These unit tests document individual refactorings and changes to resolve ambiguities.

As a side-effect of our case study, we have extended the collection of metric reference data of [15] with values for several grammars of widely used languages, and we have collected data for additional grammar metrics defined by us in [16].

Although we have used SDF as grammar formalism other formalisms such as APIs, DSLs or DSMLs might benefit from similar approach. The methodology can be used to guide the development. The importance of testing and tool support for testing is recognized in software engineering. Metrics can be used for validation, quality control, productivity and comparison.

5.2 Future Work

Lämmel generalized the notion of rule coverage and advocates the uses of coverage analysis in grammar development [21]. When adopting a transformational approach to grammar completion and correction, coverage analysis can be used to improve grammar testing, and test set generation can be used to increase coverage. SDF tool support for such test set generation and context-dependent rule coverage analysis has yet to be developed.

We plan to extend the grammar in a modular way to cover other dialects of the VDM-SL language, such as IFAD VDM and VDM⁺⁺. We have already generated Haskell support for VDM processing for the grammar, and are planning to provide generate Java support as well. The integration test suite deserves further extension in order to increase coverage.

Availability. The final version of the ISO VDM-SL grammar in SDF (development version 48, release version 0.0.3) is available as browseable hyperdocument

from <http://voodooom.sourceforge.net/iso-vdm.html>. All intermediate versions can be obtained from the CVS repository at the project web site at <http://voodooom.sourceforge.net/>, under an open source license. The SdfMetz tool used for metrics computation is available from <http://sdfmetz.googlecode.com>.

References

1. Klint, P., Lämmel, R., Verhoef, C.: Towards an engineering discipline for grammarware. *Transaction on Software Engineering and Methodology*, 331–380 (2005)
2. Malloy, B.A., Power, J.F., Waldron, J.T.: Applying software engineering techniques to parser design: the development of a *c#* parser. In: *Proc. of the 2002 Conf. of the South African Institute of Computer Scientists and Information Technologists*, pp. 75–82. In cooperation with ACM, Press, New York (2002)
3. Lämmel, R., Verhoef, C.: Semi-automatic Grammar Recovery. *Software—Practice & Experience* 31(15), 1395–1438 (2001)
4. van der Brand, M., Sellink, A., Verhoef, C.: Current parsing techniques in software renovation considered harmful. In: *IWPC 1998: Proceedings of the 6th International Workshop on Program Comprehension*, pp. 108–117. IEEE Computer Society, Los Alamitos (1998)
5. de Jonge, M., Visser, J.: Grammars as contracts. In: Butler, G., Jarzabek, S. (eds.) *GCSE 2000. LNCS*, vol. 2177, pp. 85–99. Springer, Heidelberg (2001)
6. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices* 24(11), 43–75 (1989)
7. Visser, E.: *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam (1997)
8. den van 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, p. 365. Springer, Heidelberg (2001)
9. Visser, E., Benaïssa, Z.: A Core Language for Rewriting. In: Kirchner, C., Kirchner, H. (eds.) *Proc. of the Int. Workshop on Rewriting Logic and its Applications (WRLA 1998)*, France. *ENTCS*, vol. 15. Elsevier Science, Amsterdam (1998)
10. Lämmel, R., Visser, J.: A Strafunski Application Letter. In: Dahl, V., Wadler, P. (eds.) *PADL 2003. LNCS*, vol. 2562, pp. 357–375. Springer, Heidelberg (2002)
11. Kuipers, T., Visser, J.: Object-oriented tree traversal with JJForester. In: Brand, M.v.d., Parigot, D. (eds.) *Proceedings of the Workshop on Language Descriptions, Tools and Applications (LDTA)*. *Electronic Notes in Theoretical Computer Science*, vol. 44. Elsevier, Amsterdam (2001)
12. de Jonge, M.: A pretty-printer for every occasion. In: Ferguson, I., Gray, J., Scott, L. (eds.) *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, University of Wollongong, Australia (2000)
13. Fogel, K.: *Open Source Development with CVS*. Coriolis Group Books (1999)
14. Kort, J., Lämmel, R., Verhoef, C.: The grammar deployment kit. In: van den Brand, M., Lämmel, R. (eds.) *ENTCS*, vol. 65. Elsevier, Amsterdam (2002)
15. Power, J., Malloy, B.: A metrics suite for grammar-based software. *Journal of Software Maintenance and Evolution* 16, 405–426 (2004)
16. Alves, T., Visser, J.: Metrication of SDF grammars. Technical Report DI-PUR-05.05.01, Universidade do Minho (May 2005)

17. Halstead, M.: Elements of Software Science. Operating, and Programming Systems Series, vol. 7. Elsevier, New York (1977)
18. Fenton, N., Pfleeger, S.L.: Software metrics: a rigorous and practical approach, 2nd edn. PWS Publishing Co., Boston (revised printing, 1997)
19. Bravenboer, M.: Parse Unit home page,
<http://www.program-transformation.org/Tools/ParseUnit>
20. Purdom, P.: Erratum: A Sentence Generator for Testing Parsers [BIT **12**(3), 1972, p. 372]. BIT **12**(4), 595–595 (1972)
21. Lämmel, R.: Grammar testing. In: Hussmann, H. (ed.) FASE 2001. LNCS, vol. 2029, pp. 201–216. Springer, Heidelberg (2001)
22. Alves, T., Silva, P., Visser, J., Oliveira, J.: Strategic term rewriting and its application to a VDMSL to SQL conversion. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 399–414. Springer, Heidelberg (2005)
23. International Organisation for Standardization: Information technology—Programming languages, their environments and system software interfaces—Vienna Development Method—Specification Language—Part 1: Base language, ISO/IEC 13817-1 (December 1996)
24. den van Brand, M., Scheerder, J., Vinju, J., Visser, E.: Disambiguation filters for scannerless generalized LR parsers. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, p. 143. Springer, Heidelberg (2002)
25. Lämmel, R.: The Amsterdam toolkit for language archaeology (Extended Abstract). In: Proceedings of the 2nd International Workshop on Meta-Models, Schemas and Grammars for Reverse Engineering (ATEM 2004) (October 2004)
26. Erbach, G.: Tools for grammar engineering (March 15 (2000)
27. Volk, M.: The role of testing in grammar engineering. In: Proc. of the 3rd Conf. on Applied Natural Language Processing, Assoc. for Computational Linguistics, pp. 257–258 (1992)

Sudoku – A Language Description Case Study

Terje Gjørseter, Ingelin F. Isfeldt, and Andreas Prinz

Faculty of Engineering and Science
University of Agder

Grooseveien 36, N-4876 Grimstad, Norway

terje.gjosater@uia.no, ingelin.isfeldt@gmail.com, andreas.prinz@uia.no

Abstract. A complete language description includes the structure as well as constraints, textual representation, graphical representation, and behaviour (transformation and execution). As a case study in language description, we consider Sudoku as a language, where a Sudoku puzzle is an instance of the language. Thus we are able to apply meta-model-based technologies for the creation of a language description for Sudoku, including correctness checking of a puzzle, and solving strategies. We identify what has to be expressed and how this can be done with the technology available today.

1 Introduction

To describe a computer language, whether it is a domain specific language (DSL) or a full general purpose programming language, usually involves several different technologies and meta-languages. Different language aspects such as structure, constraints, textual and graphical representation, and transformational and executional behaviour all need to be covered.

In this article, we want to identify what has to be expressed and how this can be done with the meta-model-based technology available today. In order to achieve this, we will perform a case study where we create a complete language description for a small language. This language should be easy to understand, small enough that a complete description is feasible to create within a limited time frame, and still cover all the language aspects mentioned above. We choose to create a meta-model-based language description of Sudoku, describing structure, constraints, representation as well as behaviour.

Sudoku is a very popular puzzle, and has been for some years now. The game of filling in the numbers from 1 to 9 into a 9x9 celled square of 3x3 celled sub-squares is fascinating people all over the world. Both smaller and larger Sudoku puzzles have become available, where letters and symbols are included in the game and also much more advanced problems exist, with colours, and non-square solutions. These small puzzles can be found all over the web, in newspapers and magazines; and books and computer games filled with Sudoku puzzles are widely available.

We have focussed on the essential features to be described, in order to be as abstract as possible. This description was later used as a foundation for comparative implementations in Eclipse (see [1]) and Visual Studio (see [2]). We

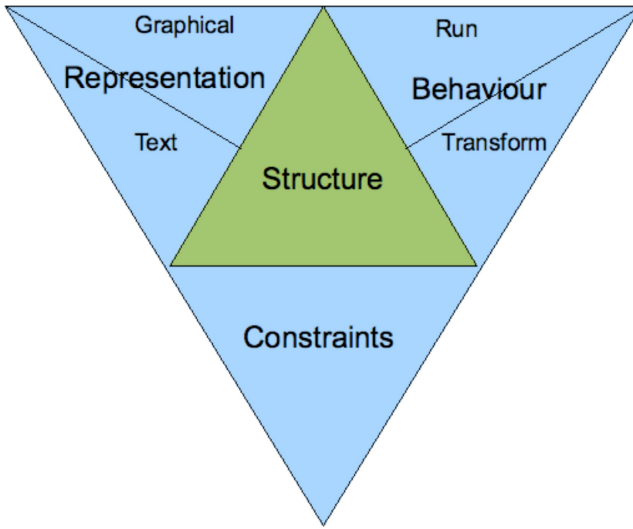


Fig. 1. Aspects of a computer language description

will focus on the platform-independent aspects here, but more details on a full implementation of Sudoku based on this language description is available in [3].

The Figure 1 below shows the components of a language description, consisting of Structure, Constraints, Representation and Behaviour, as introduced in [4].

Structure defines the constructs of a language and how they are related. This is also known as abstract syntax.

Constraints describe additional restrictions on the structure of the language, beyond what is feasible to express in the structure itself.

Representation defines how instances of the language are represented. This can be the description of a graphical or textual concrete language syntax.

Behaviour explains the dynamic semantics of the language. This can be a mapping or a transformation into another language (translational semantics, denotational semantics), or it defines the execution of language instances (operational semantics).

For a complete language description, it is necessary to not only define a meta-model that describes the structure and constraints of the language, but also to define the representation and behaviour of the language, as explained in [5], and relate these definitions to the meta-model. As the figure above suggests, all other aspects are related to the structure, and elements of the other aspects may refer to elements of the structure or be connected to elements of the structure via a mapping or a transformation.

In this article, we will attempt to create such a language description covering all of these aspects, by using Sudoku as a case.

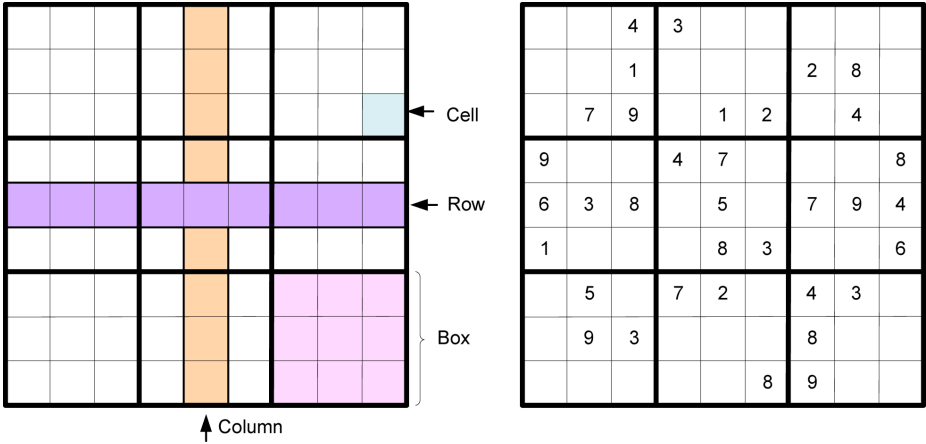


Fig. 2. An empty Sudoku (left) and a Sudoku with givens

We will endeavour to keep the language description platform-independent, and choose suitable technologies for describing each of the language aspects mentioned above. When choosing how to describe each language aspect, we have to consider what exactly it is we want to describe, and then attempting to find a suitable framework for such a description. However, in some cases such as for graphical representation, it may be difficult to achieve a suitable level of abstraction and exactness using a single technology. For some of the aspects, we will also show fragments of descriptions on a more platform-specific abstraction level, to add further details and to exemplify how the platform independent description can be taken down to an implementable level.

The rest of the article is organised as follows: Section 2 gives some background information and history on the game of Sudoku. The different aspects of language descriptions that are covered in our description of Sudoku are covered in the following sections; *structure* in Section 3, *constraints* in Section 4, *textual representation* in Section 5, *graphical representation* in Section 6, *transformation behaviour* in Section 7, and *execution behaviour* in Section 8. Finally, we draw our conclusions in Section 9.

2 Sudoku

A Sudoku usually consists of a grid with 9×9 cells divided into rows, columns and boxes. Some cells might already contain numbers, known as *givens* (see Figure 2). The goal of the game is to fill in the empty cells, one number in each cell, so that each row, column and box contains the numbers 1 to 9 exactly once. The puzzles come in several difficulties depending on the number and/or layout of the givens. We will use *field* as a common term for row, column and box.

2.1 Rules and Solving Strategies

There is essentially just one rule of solving standard Sudoku; every row, column and box must include the numbers from 1 to 9 exactly once. There are several solving strategies one can use when trying to solve a Sudoku but we will only mention a few of them here (see also [6]).

Elimination: Before trying to solve the Sudoku by using the strategies that follow, fill in all possible candidates in each cell. It is very important that when a number has been assigned to a cell, then this number is excluded as a candidate from all other cells sharing the same field. Performing this elimination will make the process of solving the Sudoku much easier as it is likely to narrow down the options for possible values in a cell.

Single candidate: If any number is the only candidate in a cell, this number must be the correct solution for this cell.

Hidden single candidate: If any number is a candidate in only one cell in a field, this number must be the correct solution for this cell.

Locked candidates: Sometimes a candidate within a box is restricted to one row or column. Since one of these cells has to contain that specific candidate, the candidate can be excluded from the remaining cells in that row or column outside of the box. Sometimes a candidate within a row or column is restricted to one box. Since one of these cells has to contain that specific candidate, the candidate can be excluded from the remaining cells in the box.

3 Structure

The structure of a language specifies what the instances of the language are; it identifies the meaningful components of each language construct, and relates them to each other, as described in [7]. 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.

For this purpose, a class diagram is well suited. We will use a MOF-based (see [8]) class diagram to describe the structure of Sudoku. In Figure 3, we see that the structure consists of *Puzzle*, *Field*, *Row*, *Column*, *Box* and *Cell*. *Puzzle* is the root model class and it contains several *Fields*. A *Field* must have references to the *Cells* that are associated with it. *Row*, *Column* and *Box* extend *Field*. A *Cell* must have an integer *iCellValue* that contains its value, while *Puzzle* has an integer *iDimension* keeping its dimension. A *Cell* must refer to exactly one referencing *Row*, *Column* and *Box*, while a *Row*, *Column* or *Box* must refer to *iDimension* *Cells*.

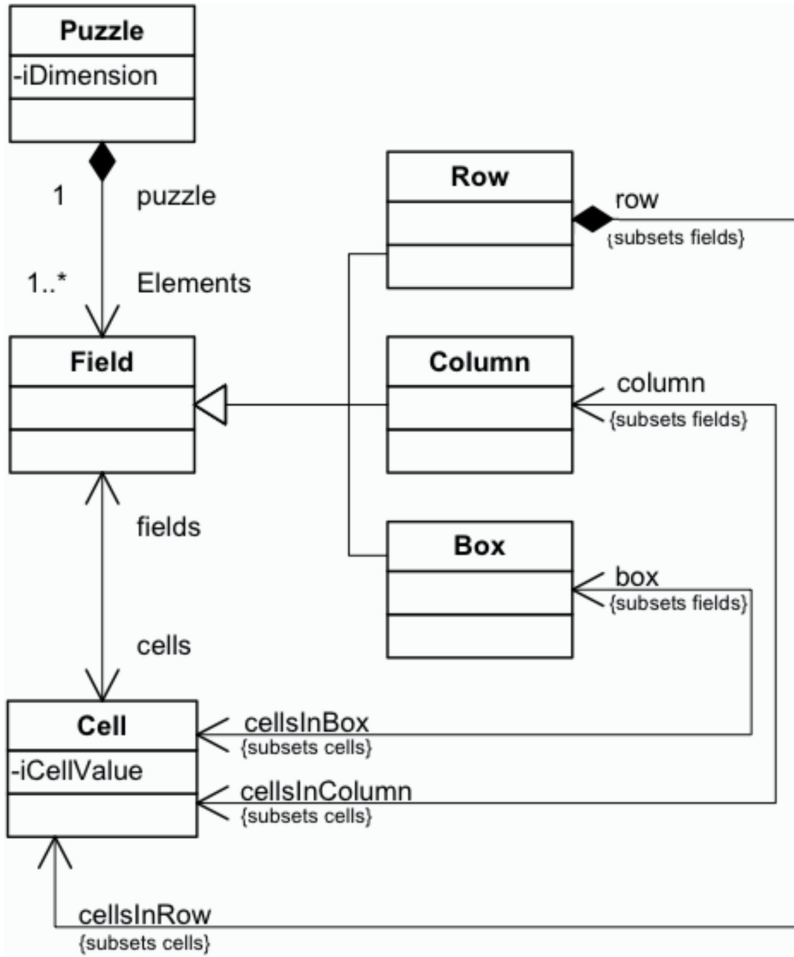


Fig. 3. A structure meta-model for Sudoku

4 Constraints

Constraints on a language can put limitations on the structure of a well-formed instance of the language. What we want to express with constraints are logical expressions. This aspect of a language description mostly concerns rules or constraints on the structure that are difficult to express directly in the structure itself. The constraints could be first-order logical constraints or multiplicity constraints for elements of the structure (see [4]).

The structure and constraints aspects of a language can sometimes overlap. Some restrictions may obviously belong to one of them, but many could belong to either of them, depending on choice or on the expressiveness of the technology used to define the structure. For our case, we use constraints for restrictions

that are difficult or impossible to model as part of the structure in the form of a MOF-based class diagram.

Constraints are a valuable help for model validation. The structure defined in Section 3 does not provide all necessary limitations that this project demands. For example, we need to make sure that a Row has exactly `iDimension` cells.

In addition to constraints on the structure, constraints on representation and behaviour (semantic constraints) may also be needed. Constraints on representation may for example be well-formedness-rules, or rules for layout and white-space usage. Behavioural constraints may for example concern elements that only exist in runtime, or may only be valid at certain stages of execution.

The Object Constraint Language (OCL, see [9]) is a formal language for describing expressions on UML models, and is thus well suited for our purpose. These expressions typically specify invariants, pre- and post- conditions that must hold for the system being modelled, and OCL can also express queries over objects described in a model. When OCL expressions are evaluated, they do not have side effects. This means that their evaluation cannot change the state of the executed system.

There are two questions that are not covered in the OCL language that are also important to consider:

- When should the constraint be evaluated? Some constraints must be valid at all times, and some are only relevant when we think that we may have solved the puzzle and want to check that it is correctly filled out.
- How serious it is to break a constraint? Is it acceptable if the constraint is temporarily broken during the lifetime of the Puzzle?

These issues can be dealt with in the implementation, for example by customising the interface to the constraint checking engine.

The following constraints are expressed in OCL and apply for a 9x9 Sudoku. The constraints *PuzzleDimension*, *RowBoxCommonCells* and *ValueCellValue* are also given as logical expressions to help readers that are not familiar with OCL.

4.1 Constraints on the Puzzle

For the whole puzzle, we have defined the following constraints:

PuzzleDimension: The Puzzle dimension must be 9.

```
context Puzzle inv PuzzleDimension:
self.iDimension = 9
```

Corresponding logical expression: $\forall p \in \text{Puzzle} \bullet iDimension(p) = 9$

PuzzleRow: A Puzzle must have as many rows as the puzzle `iDimension`.

```
context Puzzle inv PuzzleRow:
self.Elements->select(f : Field | f.ocIsTypeOf(Row))
-> size()=iDimension
```

PuzzleColumn: A Puzzle must have as many columns as the Puzzle iDimension. The OCL is similar to the previous.

PuzzleBox: A Puzzle must have as many boxes as the Puzzle iDimension. The OCL is similar to the previous.

4.2 Constraints on Fields

For fields of the puzzle, the following constraints are defined:

CellsInField: A Field must have as many cells as the Puzzle dimension.

```
context Field inv CellsInField:
self.cells -> size() = iDimension
```

RowColumnCommonCells: A Row and a Column can only have exactly one Cell in common.

```
context Row inv RowColumnCommonCells:
self.row.cells -> intersection(self.column.cells)->size() = 1
```

RowBoxCommonCells: A Row and a Box must have either 0 or exactly square root of iDimension cells in common.

```
context Cell inv RowBoxCommonCells:
self.row.cells -> intersection(self.box.cells)->size() = 0 or
self.row.cells -> intersection(self.box.cells)->size()
= square_root(iDimension)
```

Corresponding logical expression:

$$\forall c \in \text{Cell} \bullet \#(\text{cells}(\text{row}(c)) \cap \text{cells}(\text{box}(c))) \in \{0, \sqrt{iDimension(p)}\}$$

ColumnBoxCommonCells: A Column and a Box must have either 0 or exactly square root of iDimension cells in common. The OCL is similar to the previous.

4.3 Constraints on Cells

Cells of a Sudoku puzzle have the following constraints:

ValueCellValue: The Cell.iCellValue must have a value from 0 to iDimension (0 meaning the Cell is empty).

```
context Cell inv ValueCellValue:
self -> forAll(iCellValue <= iDimension and iCellValue >= 0)
```

Corresponding logical expression: $\forall c \in \text{Cell} \bullet 0 \leq i\text{CellValue}(c) \leq 9$

CellRowReference: Each Cell must only reference exactly one Row.

```
context Cell inv CellRowReference:
self.row -> size()=1
```

CellColumnReference: Each Cell must only reference exactly one Column. The OCL is similar to the previous.

CellBoxReference: Each Cell must only reference exactly one Box. The OCL is similar to the previous.

CommonRowColumn: Two cells can only have row or column in common.

```
context Cell inv CommonRowColumn:
self -> forAll(c: Cell | self<>c implies
(self.row != c.row or self.column!= c.column))
```

4.4 Semantic Constraints

Finally, we have a constraint that is not adding restrictions on the general structure, but rather on the behaviour of the puzzle:

UniqueValues: All Cell.iCellValues in one field must be unique. This constraint is only relevant when the Puzzle is considered to be solved. Before that, the cells that are not filled out would break this constraint.

```
context Field inv UniqueValues:
self.cells->self.cells->isUnique(cell : Cell | cell.iCellValue)
```

5 Textual Representation

The representation 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 be used in the language, what words have a 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 usually expressed in a grammar for textual languages.

The grammar for a textual editor for Sudoku should be quite simple and straight forward, letting the user create a Sudoku by filling *Rows* with *Cells* simply by writing text. A grammar for the textual representation can be expressed like this in EBNF (Extended Backus-Naur Form):

```
Puzzle ::= "Puzzle" "(" iDimension ")" "=" Field;
Field ::= (Row ",")* Row;
Row ::= "Row" "(" Cells ")";
Cells ::= (iCellValue ",")* iCellValue;
iDimension ::= INT;
iCellValue ::= INT;
```

By using Eclipse with the TEF (Textual Editing Framework) plug-in (see [10]), the same grammar could be expressed like this (in TEF version 0.5.0):

```

syntax toplevel PuzzleTpl,
    ecorepath "platform:
        /resource/Sudoku/resources/newstructure.ecore" {
    element CellTpl for Cell{
        single for iCellValue, with INTEGER;
    }
    element RowTpl for Row{
        "Row"; "(";
        sequence for
            cellsInRow, with @CellTpl, separator ",", last false;
        ")";
    }
    element PuzzleTpl for Puzzle{
        "Puzzle"; "(";
        single for iDimension, with INTEGER;
        ")";
        "=";
        sequence for
            Elements, with @FieldTpl, separator ",", last false;
    }
    choice FieldTpl for Field{
        @RowTpl
    }
}

```

In the TEF grammar, the *ecorepath* defines the path to the meta-model that is defined in an ecore file. We then define the non-terminals *PuzzleTpl*, *FieldTpl*, *RowTpl* and *CellTpl* as corresponding to the meta-model elements *Puzzle*, *Field*, *Row* and *Cell*.

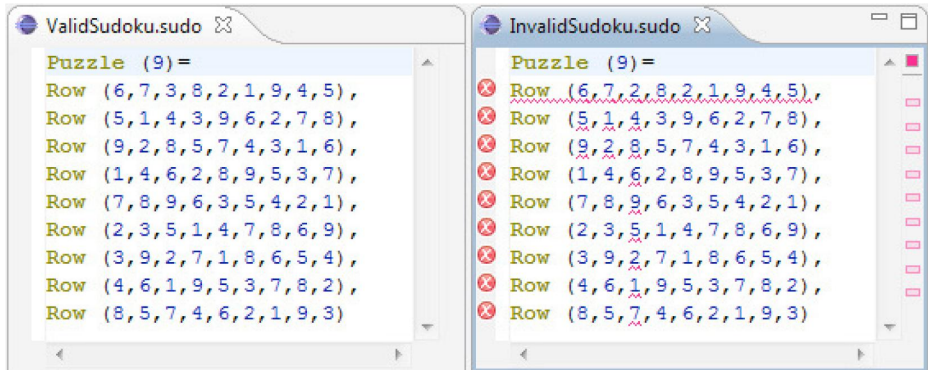


Fig. 4. A valid (left) and invalid (right) Sudoku in Eclipse with TEF

Both the EBNF grammar and the TEF grammar show a puzzle as consisting of rows limited by '(' and ')', that again consist of cells with numerical value, using ',' as separator between cells. Only *Row* and *Cell* need to be explicitly represented by textual elements, the other elements of the structure should be handled automatically. Figure 4 shows a valid and an invalid Sudoku puzzle in an editor created from this grammar in the TEF-based editor. The invalid puzzle has a row, a column and a box where the number '2' occurs twice. This breaks the constraint *UniqueValues: All Cell.iCellValues in one field must be unique*. The broken constraint is visualised in the editor by underlining all cells in the invalid fields with a red line.

6 Graphical Representation

For a graphical language, the representation will express what different symbols are used in the language, what they look like, and how they can be connected and modified to form a meaningful unit in the language. While a grammar is an obvious choice for defining textual representation, there is to our knowledge no similarly obvious choice for graphical representation available on a platform independent level among the meta-modelling standards and related technologies. A class diagram is able to show how the graphical elements are connected, but not the location and appearance of the elements. We will therefore in this case use a class diagram, but supplement it with an example implementation.

For the graphical representation of Sudoku, we want editors that let a user edit a graphically presentable Sudoku puzzle. Ideally when the user starts a new Sudoku editor, she should be presented with a ready-to-use Sudoku that contains the correct number of *Rows* containing *Cells*, *Columns* and *Boxes*. All references must be handled automatically. Figure 5 shows a model of the graphical representation. Like for the textual representation, only *Row* and *Cell* need to be explicitly represented by graphical elements, the other elements of the structure should be handled automatically.

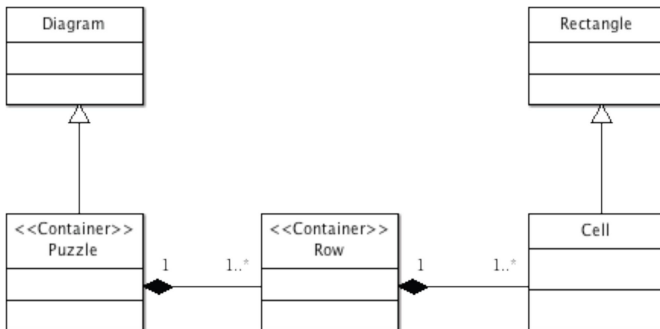


Fig. 5. A graphical representation model for Sudoku

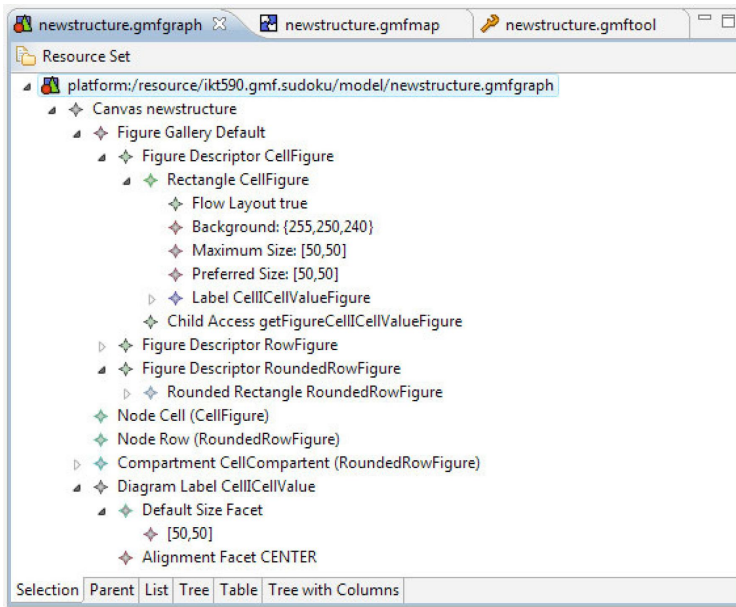


Fig. 6. A description of a graphical editor in GMF based on the graphical representation model for Sudoku

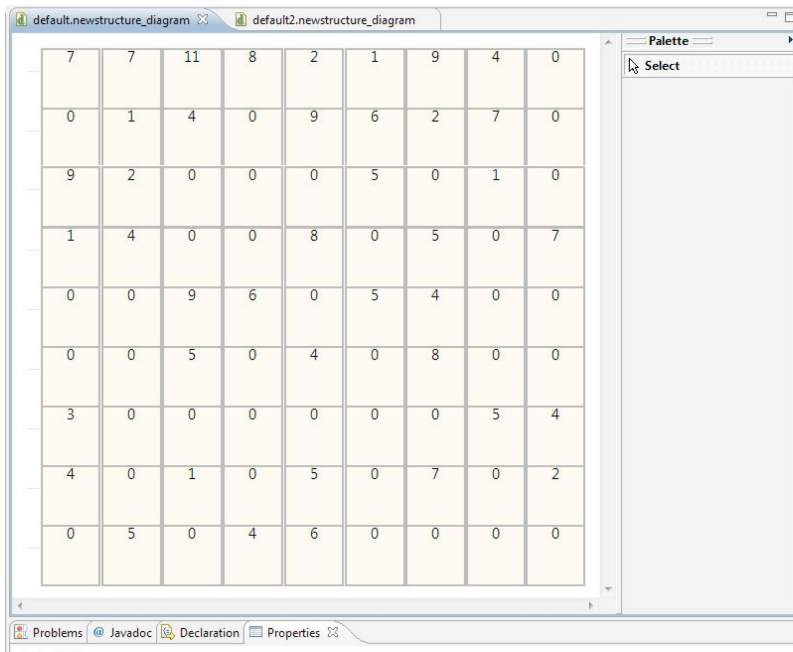


Fig. 7. A graphical editor based on the graphical representation model for Sudoku

Figure 6 shows a graphical editor description from an implementation of the graphical representation using Eclipse with the GMF plug-in (Graphical Modelling Framework - see [11]). GMF allows the creation of graphical elements and the mapping of these graphical elements to model elements. This lets the developer create graphical editors for their models. GMF takes a meta-model defined in EMF/ECORE (see [12]) as input and combines this model with graphic and tool definitions to create a mapping definition and then a generator model that generates an editor/diagram plug-in based on the provided model and definitions.

The graphical editor created from this description is shown in Figure 7.

7 Transformation Behaviour

Kleppe et al. define a transformation as “the automatic generation of a target model from a source model, according to a transformation definition” in [13].

There are two types of transformations that are particularly interesting for our case, namely model-to-text transformations and model-to-model transformations.

In model to text transformations, a model conforming to a given meta-model, is converted into a textual representation of a given meta-model that may be the same as the first meta-model, or different. This can for example be used for generating Java code from a diagram.

In model-to-model transformations, a model conforming to a given meta-model is converted into another model conforming to a given meta-model that may be the same as the first, or a different one.

One suitable option for describing model-to-model transformations is QVT (Query/View/Transform, see [14]) from the Object Management Group (OMG).

Since code is well understood, it may also be useful to describe a transformation using code. The most suitable form for code for this usage is ASM (Abstract State Machines, see [15]), because of it’s high level of abstraction.

The most interesting type of transformation for a Sudoku puzzle is a model-to-model transformation to generate more valid Sudoku puzzles from an existing puzzle. There are several such transformations that can be performed on a Sudoku puzzle without altering the logic:

- Permutations of rows and columns within blocks.
- Permutations of block rows and block columns. A block refers to a row or column of boxes, e.g. Row 1, 2 and 3 is a block row and Column 4, 5 and 6 form a block column.
- Permutations of the symbols used in the board.
- Transposing the Sudoku.

Two of these transformations will be further investigated; transposition, and permutations of the symbols used in the board.

The matrix transpose, most commonly written M^T , is the matrix obtained by exchanging rows and columns of M . Stated differently, given an $m \times n$ matrix

M , the transpose of M is the $n \times m$ matrix denoted by M^T whose columns are formed from the corresponding rows of $M[6]$:

$$M_{ij}^T = M_{ji} \text{ for } 1 \leq i \leq n, 1 \leq j \leq m \quad (1)$$

Transpose with ASM:

```
Transpose(s1,s2:Puzzle) = def
  forall i in [0..9] do
    forall j in [0..9] do
      s2.rows[i].cell[j].iCellValue
      := s1.rows[j].cell[i].iCellValue
```

The symbols used are the numbers from 1 to 9. By permutation we can arrange the Sudoku in such a way that the cells in the first row are ordered ascending from 1 to 9. In order to achieve this, the *iCellValue*s from the first *Row* must be retrieved, and then used to switch all the *iCellValue*s in the *Puzzle*. E.g. if the first *Cell* in the first *Row* has *iCellValue* = 6, then all *Cells* where *iCellValue* = 6 must be changed to *iCellValue* = 1.

Using the QVT Relations language (see [14]), sorting of the first row may be described as the code example below. Note that the cell values are temporarily changed to the intended final value + 10 to avoid already changed values to be changed again during the next steps.

```
transformation SortFirstRow(source : newstructure,
                             target: newstructure){
top relation getNinthValue {
  row1:Field;
  checkonly domain source puzzle:Puzzle
  {};
  checkonly domain source cell:Cell
  {iCellValue = row1.cells->at(9).iCellValue};
  enforce domain target cell:Cell
  {iCellValue = 19};
  when
  {row1 = puzzle.Elements
   ->select(f:Field|f.oclIsTypeOf(Row))->first();}
}
top relation getEightValue {
  row1:Field;
  checkonly domain source puzzle:Puzzle
  {};
  checkonly domain source cell:Cell
  {iCellValue = row1.cells->at(8).iCellValue};
  enforce domain target cell:Cell
  {iCellValue = 18};
  when
  {row1 = puzzle.Elements
```



```

        ->select(f:Field|f.oclIsTypeOf(Row))->first();}
    }
--Same procedure followed for all values, code omitted

}}
```

8 Execution Behaviour

For operational semantics, execution can be described with the following 3 elements:

1. **Runtime environment**, i.e. *runtime state space*. This can be expressed as an extension of the meta-model, with runtime elements added to express possible states (see [16]).
2. **Initial state** of the system, this can be described as an object diagram, or simply with constraints.
3. **State transitions**, i.e. *code*. The most abstract way to express code is using ASM, but QVT transformations can also be used for defining state transitions since they can be seen as a series of transformations between different states. In the case of Sudoku we may also have based the execution on a structure editor backed by an incrementally evaluating attribute grammar.

The strategies that we have defined are the ones listed in Section 2.1. In order to execute these solving strategies, we need to define a list *iPossibleCellValues* to keep track of the values that are available in a *Cell* at any given time during the solving process. Figure 8 shows the runtime metamodel for Sudoku, where *Cell* is extended with a *RTCell* with the added *iPossibleCellValues* list.

The initial state can then be defined using OCL, by defining the initial values of *iPossibleCellValues*:

```

context RTCell
    iPossibleCellValues = {1,2,3,4,5,6,7,8,9}
```

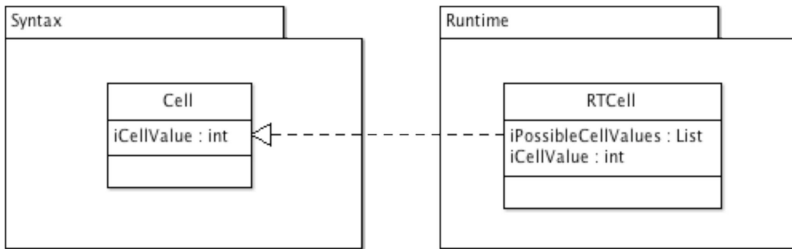


Fig. 8. A runtime meta-model for Sudoku

The following ASM code describes state transitions of an execution of the listed Sudoku solving strategies:

```

Run(s:Sudoku) =def
  forall f in s.field do RunElimination (f)
  forall f in s.field do RunSingleCell (f)
  forall f in s.field do RunHiddenCell (f)
  forall f in s.field do RunLockedCandidates (f)

RunElimination(f:Field) =def
  forall c in f.cells with c.iCellValue<>0 do
    forall cc in f.cells do
      delete c.iCellValue from cc.iPossibleCellValues

RunSingleCell(f:Field) =def
  forall c in f.cells with c.iCellValue = 0
    and c.iPossibleCellValues.size = 1 do
    choose v in c.iPossibleCellValues do c.iCellValue := v

RunHiddenCell(f:Field) =def
  forall v in [1..iDimension] do
    let possibleCells
      = { c in f.cells with v in c.iPossibleCellValues}
    if possibleCells.size = 1 then
      choose c in possibleCells do
        c.iCellValue := v

RunLockedCandidates(f:Field) =def
  forall otherF in puzzle.field with otherF <> f
    and (f.cell intersect otherF.cell).size > 1 do
    forall v in [1..iDimension] do
      let possibleCells
        = { c in f.cells with v in c.iPossibleCellValues}
      if possibleCells subsetof otherF.cells
        forall cc in otherF.cells with cc not in f.cells
          delete v from cc.iPossibleValues

```

For single cell candidate, the *RunSingleCell* function above could also be expressed like this in QVT Relations:

```

transformation Solve(source: newstructure, target: newstructure){
  top relation solveCell{
    checkonly domain source cell:Cell{
      iCellValue = 0};
    enforce domain target cell:Cell{
      iCellValue = iPossibleValues->at(0)};
  }

```

```

where{
    iPossibleValues->size()==1};
}

```

9 Conclusions

We have described how we have created a complete language description with state-of-the-art meta-model-based technologies, using the puzzle game of Sudoku as an example. We have attempted to select suitable technologies for each language aspect that was listed in Section 1, on a sufficiently high level of abstraction to be platform independent, and at the same time with a sufficient level of formality to be unambiguous.

We are satisfied with Sudoku as a test case for language description, it has allowed us to develop a language description that covers all the important language aspects; structure, constraints, representation and behaviour. We have used a MOF-based class diagram to define the structure and OCL for the constraints. For the textual representation, we have defined an EBNF grammar, and also shown how this can be used in an implementation with the TEF editor creation framework. Graphical representation is described by a class diagram displaying the intended relationship between model elements, and supplemented with an example implementation using the GMF framework. Transformation behaviour is primarily described using QVT, and execution is primarily described using ASM. For most of these language aspects, the chosen technology has been adequate for the task. For structure, constraints and textual representation, we have been able to describe the aspect on a suitable level of abstraction. For graphical representation, execution behaviour and transformation, the choice of technology was not so obvious. The weakest point is the graphical representation, where the chosen technology was capable of describing the structure of the representation elements, but not the layout and appearance. This was therefore mitigated by adding a reference implementation using GMF.

In this way, we have managed to create a complete description of Sudoku that has been successfully used as a starting point for more detailed descriptions in Eclipse with various plug-ins, as well as in Visual Studio with DSL-tools, as described in [3].

References

1. d'Anjou, J., Fairbrother, S., Kehn, D., Kellermann, J., McCarthy, P.: The Java Developer's Guide to Eclipse. Addison-Wesley, Reading (2004)
2. Microsoft: Introducing Visual Studio, <http://msdn2.microsoft.com/en-us/library/6x6bk1f4VS.80.aspx>
3. Isfeldt, I.F.: A metamodel for sudoku. Master's thesis, University of Agder (2008), <http://student.grm.hia.no/master/ikt07/ikt590/g01>
4. 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)

5. Kleppe, A.: A language is more than a metamodel. In: ATEM 2007 workshop (2007), <http://megaplanet.org/atem2007/ATEM2007-18.pdf>
6. Delahaye, J.P.: The science behind sudoku. *Scientific American*, 80–87 (June 2006)
7. Sethi, R.: *Programming Languages Concepts and Constructs*. Addison-Wesley, Reading (1996)
8. OMG (ed.): Revised Submission to OMG RFP ad/2003-04-07: Meta Object Facility (MOF) 2.0 Core Proposal. Technical report, Object Management Group (April 2003), <http://www.omg.org/docs/formal/06-01-01.pdf>
9. OMG: OCL 2.0 Specification. Object Management Group (June 2005)ptc/2005-06-06
10. Scheidgen, M.: Textual Editing Framework,
<http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/documentation.html>
11. GMF developers: Eclipse Graphical Modeling Framework,
<http://www.eclipse.org/gmf>
12. Griffin, C.: Using EMF. Technical report, IBM: Eclipse Corner Article (2003), <http://www.eclipse.org/articles/Article-UsingEMF/using-emf.html>.
13. Kleppe, A., Warmer, J., Bast, W.: *MDA explained: the model driven architecture: practice and promise*. Object Technology Series. Addison – Wesley, Reading (2003)
14. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Final Adopted Specification ptc/05-11-01. OMG document, Object Management Group (2005), <http://www.omg.org/docs/ptc/05-11-01.pdf>
15. Börger, E., Stärk, R.: *Abstract State Machines*. Springer, Heidelberg (2003)
16. Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. In: *European Conference on Model Driven Architecture: Foundations and Applications* (2007)

The Java Programmer’s Phrase Book

Einar W. Høst and Bjarte M. Østvold

Norwegian Computing Center
{einarwh,bjarte}@nr.no

Abstract. Method names in Java are natural language phrases describing behaviour, encoded to make them easy for machines to parse. Programmers rely on the meaning encoded in method names to understand code. We know little about the language used in this encoding, its rules and structure, leaving the programmer without guidance in expressing her intent. Yet the meaning of the method names — or phrases — is readily available in the body of the methods they name. By correlating names and implementations, we can figure out the meaning of the original phrases, and uncover the rules of the phrase language as well. In this paper, we present an automatically generated proof-of-concept phrase book for Java, based on a large software corpus. The phrase book captures both the grammatical structure and the meaning of method phrases as commonly used by Java programmers.

1 Introduction

Method identifiers play three roles in most programming languages. The first is a technical one: method identifiers are *unique labels* within a class; strings of characters that act as links, allowing us to unambiguously identify a piece of code. If we want to invoke that piece of code, we refer to the label. The second role is mnemonic. While we could, in theory, choose arbitrary labels for our methods, this would be cumbersome when trying to remember the correct label for the method we want to invoke. Hence methods are typically given labels that humans can remember, leading us to refer to method identifiers also as method *names*. Finally, method identifiers play a semantic role. Not only do we want labels we can remember; we want them to express meaning or intent. This allows us to recall what the method actually does. Unfortunately, we lack an established term for this role — identifiers are not just names. Rather, they are structured expressions of intent, composed of one or more fragments. Indeed they are *method phrases*, utterances in natural language, describing the behaviour of methods.

Consider, for instance, the following example, defining the Java method `findElementByID`:

```
Element findElementByID(String id) {  
    for (Element e : this.elements) {  
        if (e.getID().equals(id)) {
```

```

        return e;
    }
}
return null;
}

```

We immediately observe that the form of the method phrase is somewhat warped and mangled due to the hostile hosting environment. Since the phrase must double as a unique label for the method, and to simplify parsing, the phrase must be represented by a continuous strings of characters. But it is nevertheless a phrase, and we have no problems identifying it as such. In a more friendly environment, we would unmanacle the phrase and simply write *Find element by ID*. For instance, in the programming language Subtext¹, the roles as links and names are completely decoupled: the names are mere comments for the links. This leaves the programmer with much greater flexibility when naming methods. Edwards argues that “names are too rich in meaning to waste on talking to compilers” [1].

Looking at the implementation, we see that there are several aspects that conspire to match the expectations given us by the phrase: the method returns a reference to an object, accepts a parameter, contains a loop, and has two return points. We posit that all meaningful method phrases can similarly be described, simply by noting the distinguishing aspects of their implementations. Our goal is to automatically generate such descriptions for the most common method phrases in Java.

Since the phrase and the implementation of a method should be in harmony, we cannot arbitrarily change one without considering to change the other. We want the phrase to remain a correct abstract description of the implementation of the method, otherwise the programmer is lying! Therefore, if the implementation is changed, the phrase should ideally be changed to describe the new behaviour². Conversely, if the phrase is changed, care should be taken to ensure that the implementation fulfills the promise of the new phrase. Unfortunately, programmers have no guidance besides their own intuition and experience to help make sure that their programs are truthful in this sense. In particular, programmers lack the ability to assess the quality of method phrases with regards to suitability, accuracy and consistency.

Realizing that method names are really phrases, that is, expressions in natural language, allows us some philosophical insight into the relationship between the method name and the method body or implementation. Frege distinguishes between the *sign* — name, or combination of words —, the *reference* — the object to which the sign refers — and the *sense* — our collective understanding of the reference [2]. Note that the sense is distinct from what Frege calls the *idea*, which is the individual understanding of the reference. Depending on the

¹ <http://subtextual.org>

² However, since phrases act as links, this is not always practical: changing phrases in public APIs may break client code.

insight of the individual, the idea (of which there are many) may be in various degrees of harmony or conflict with the sense (of which there is only one).

In this light, the creation of a method is a way of expression where the programmer provides both the sign (the method name) and a manifestation of the idea (the implementation). The tension between the individual idea and the sense is what motivates our work: clearly it would be valuable to assist the programmer in minimizing that tension. Understanding the language of method phrases is a first step towards providing non-trivial assistance to programmers in the task of naming. In the future, we will implement such assistance in a tool.

We have previously shown how to create a semantics which captures our common interpretation, or sense, of the *action verbs* in method names [3]. Building on this work, we use an augmented model for the semantics, and expand from investigating verbs to full method names, understood as natural language phrases.

The main contributions of this paper are as follows:

- A perspective on programming that treats method names formally as expressions in a restricted natural language.
- The identification of a restricted natural language, *Programmer English*, used by Java programmers when writing the names of methods (Section 2.1).
- An approach to encoding the semantics of methods (Section 3.2), expanding on our previous work.
- An algorithm for creating a relevant and useful phrase book for Java programmers (Section 4.2).
- A proof-of-concept phrase book for Java that shows the potential and practicality of our approach (see Section 5 for excerpts).

2 Conceptual Overview

Our goal is to describe the meaning and structure of method names as found in “the real world” of Java programming, and present the findings in a *phrase book* for programmers. Our approach is to compare method names with method implementations in a large number of Java applications. In doing so, we are inspired by Wittgenstein, who claimed that “the meaning of a word is its use in the language” [4]. In other words, the meaning of natural language expressions is established by pragmatically considering the contexts in which the expressions are used.

2.1 Programmer English

Method names in Java are phrases written in a natural language that closely resembles English. However, the language has important distinguishing characteristics stemming from the context in which it is used, affecting both the grammar and the vocabulary of the language.

The legacy of short names lingering from the days before support for automatic name completion still influences programmers. This results in abbreviations and degenerate names with little grammatical structure. While increasing focus on readability might have improved the situation somewhat, Java is

still haunted by this culture. A recent example is the `name` method defined for *enums*, a language feature introduced in Java 5.0. A more explicit name would be `getName`.

Futhermore, the vocabulary is filled with general computing terms, technology acronyms, well-known abbreviations, generic programming terms and special object-oriented terms. In addition, the vocabulary of any given application is extended with domain terms, similar to the use of foreign words in regular English. This vocabulary is mostly understandable to programmers, but largely incomprehensible to the English-speaking layman.

We therefore use the term *Programmer English* to refer to the special dialect of English found in Java method names. Of course, Programmer English is really *Java Programmer English*, and other “Programmer Englishes” exist which might exhibit quite different characteristics. For instance, *Haskell Programmer English* is likely to be radically different, whereas *Ruby Programmer English* probably shares some traits with Java Programmer English, and *C# Programmer English* is likely to be near-identical.

2.2 Requirements for the Phrase Book

The main requirements for a phrase book is that it be *relevant* and *useful*. For the phrase book to be relevant, it must stem from “the trenches” of programming. In other words, it must be based on real-world data (i.e. programs), and be representative of how typical Java programmers express themselves.

The usefulness requirement is somewhat more subtle: what does it mean to be useful? Certainly, the phrase book should have a certain amount of *content*, and yet be *wieldy*, easy to handle for the reader. Hence, we want to be able to adjust the number of phrases included in the phrase book. In addition, each phrase must be useful in itself. We propose the following three requirements to ensure the usefulness of phrases: 1) each phrase must have a description that matches actual usage, 2) each phrase must have a well-understood semantics, and 3) each phrase must be widely applicable. These are requirements for *validity*, *precision* and *ubiquity*, respectively.

Since each Java application has its own specialized vocabulary, we must be able to abstract away domain-specific words. The phrase book should therefore contain both concrete phrases such as `get-last-element` and abstract ones such as `find-[noun]`. We prefer concrete phrases since they are more directly applicable, but need abstract phrases to fill out the picture.

We also decide that the phrase book should be organized hierarchically, as a tree. That way, the phrase book directly reflects and highlights the grammatical structure of the phrases themselves. This also makes the phrase book easier to browse. The phrases should therefore be organized as refinements of each other. Since a phrase represents a set of methods, its refinements are a partitioning of the set. Note that the partitioning is syntax-driven: we cannot choose freely which methods to group together. At any given step, we can only choose refined phrases supported by the grammar implicitly defined by the corpus.

2.3 Approach

Figure 1 provides an overview of our approach. The analysis consists of two major phases: data preparation and phrase book generation.

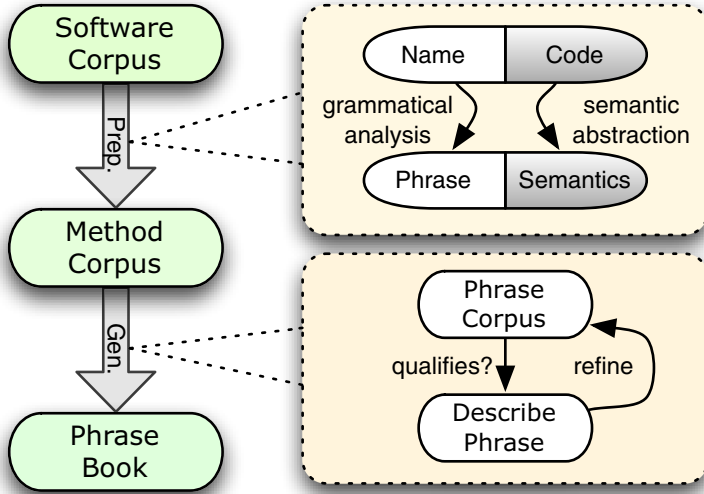


Fig. 1. Overview of the approach

In the preparation phase, we transform our corpus of Java applications into an idealized corpus of methods. Each Java method is subject to two parallel analyses. On one hand, we analyze the grammatical structure of the method name. This analysis involves decomposing the name into individual words and the part-of-speech tagging of those words. This allows us to abstract over the method names and create phrases consisting of both concrete words and abstract categories of words. On the other hand, we analyze the bytecode instructions of the implementations, and derive an abstract semantics. We can then investigate the semantics of the methods that share the same phrase, and use this to characterize the phrase.

This is exactly what happens in the generation phase. We apply our recursive phrase book generation algorithm on the corpus of methods. The algorithm works by considering the semantics of gradually more refined phrases. The semantics of a phrase is determined by the semantics of the individual methods with names that match the phrase. If a phrase is found to be useful, it receives a description in the phrase book. We generate a description by considering how the phrase semantics compares to that of others. We then attempt to refine the phrase further. When a phrase is found to be not useful, we terminate the algorithm for that branch of the tree.

2.4 Definitions

We need some formal definitions to be used in the analysis of methods (Sect. 3) and when engineering the phrase book (Sect. 4). First, we define a set \mathcal{A} of attributes that each highlight some aspect of a method implementation. An *attribute* $a \in \mathcal{A}$ can be evaluated to a binary value $b \in \{0, 1\}$, denoting absence or presence. A *method* m has three features: a unique *fingerprint* u , a *name* n , and a list of *values* b_1, \dots, b_n for each $a \in \mathcal{A}$. These values are the *semantics* of the method. Unique fingerprints ensure that a set made from arbitrary methods m_1, \dots, m_k always has k elements. The name n consists of one or more *fragments* f . Each fragment is annotated with a *tag* t .

A *phrase* p is an abstraction over a method name, consisting of one or more *parts*. A part may be a fragment, a tag or a special wildcard symbol. The wildcard symbol, written $*$, may only appear as the last part of a phrase. A phrase that consists solely of fragments is *concrete*; all other phrases are *abstract*. A concrete phrase, then, is the same as a method name.

A phrase *captures* a name if each individual part of the phrase captures each fragment of the name. A fragment part captures a fragment if they are equal. A tag part captures a fragment if it is equal to the fragment's tag. A wildcard part captures any remaining fragments in a name. A concrete phrase can only capture a single name, whereas an abstract phrase can capture multiple names. For instance, the phrase **[verb]-valid-*** captures names like **is-valid**, **has-valid-signature** and so forth. The actual set of captured names is determined by the corpus.

A *corpus* \mathcal{C} is a set of methods. Implicitly, \mathcal{C} defines a set \mathcal{N} , consisting of the names of the methods $m \in \mathcal{C}$. A *name corpus* \mathcal{C}_n is a set of methods with the name n . A *phrase corpus* \mathcal{C}_p is a set of methods whose names are captured by the phrase p . The *relative frequency value* $\xi_a(\mathcal{C})$ for an attribute a given a corpus \mathcal{C} is defined as:

$$\xi_a(\mathcal{C}) \stackrel{\text{def}}{=} \frac{\sum_{m \in \mathcal{C}} b_a(m)}{|\mathcal{C}|},$$

where $b_a(m)$ is the binary value for the attribute a of method m . The semantics of a corpus is defined as the list of frequency values for all $a \in \mathcal{A}$, $[\xi_{a_1}(\mathcal{C}), \dots, \xi_{a_m}(\mathcal{C})]$. We write $\llbracket p \rrbracket$ for the semantics of a phrase, and define it as the semantics of the corresponding phrase corpus.

If two methods m, m' have the same values for all attributes we say that they are *attribute-value identical*, denoted $m \simeq m'$. Using relation \simeq we can divide a corpus \mathcal{C} into a set of equivalence classes $\text{EC}(\mathcal{C}) = [m_1]_{\mathcal{C}}, \dots, [m_k]_{\mathcal{C}}$, where $[m]_{\mathcal{C}}$ is defined as:

$$[m]_{\mathcal{C}} \stackrel{\text{def}}{=} \{m' \in \mathcal{C} \mid m' \simeq m\}.$$

We simplify the notation to $[m]$ when there can be no confusion about the interpretation of \mathcal{C} . Now we apply some information-theoretical concepts related to entropy [5]. Let the probability mass function $p([m])$ of corpus \mathcal{C} be defined as:

$$p([m]) \stackrel{\text{def}}{=} \frac{|[m]|}{|\mathcal{C}|}, \quad [m] \in \text{EC}(\mathcal{C}).$$

We then define the *entropy* of corpus \mathcal{C} as:

$$H(\mathcal{C}) \stackrel{\text{def}}{=} H(p([m_1]), \dots, p([m_k])).$$

Finally, we define the entropy of a phrase as the entropy of its phrase corpus.

3 Method Analysis

When programmers express themselves in Programmer English, they give both the actual expression (the name) and their subjective interpretation of that expression (the implementation). We therefore analyze each method in two ways: (a) a syntactic analysis concerned with interpreting the name, and (b) a semantic analysis concerned with interpreting the implementation. The input to the analyses is a Java method as found in a Java class file, the output an idealized *method* as defined in Sect. 2.4, consisting of fingerprint, name and semantics.

3.1 Syntactic Analysis of Method Names

Method names are not arbitrarily chosen; they have meaning and structure. In particular, names consist of one or more *words* (which we call fragments) put together to form phrases in Programmer English. We apply natural language processing techniques [6], in particular *part-of-speech tagging*, to reveal the grammatical structure of these phrases.

Decomposition of Method Names. Since whitespace is not allowed in identifiers, the Java convention is to use the transition from lower-case to upper-case letters as delimiter between fragments. For example, a programmer expressing the phrase “get last element” would encode it as `getLastElement`. Since we want to analyze the method names as natural language expressions, we reverse engineer this process to recover the original phrase. This involves *decomposing* the Java method names into fragments.

Since we are focussed on the typical way of expression in Java, we discard names that use any characters except letters and numbers. This includes names using underscore rather than case transition as a delimiter between fragments. Since less than 3% of the methods in the original corpus contain underscores, this has minimal impact on the results. That way, we avoid having to invent ad-hoc rules and heuristics such as *delimiter precedence* for handling method names with underscores *and* case transition. For instance, programmers may mix delimiters when naming test methods (e.g., `test_handlesCornerCase`), use underscores for private methods (e.g., `_findAccount`) or in other special cases (e.g., `processDUP2_X2`). Indeed, nearly half the names containing underscores have an underscore as the first character.

Part-of-speech Tagging. The decomposed method name is fed to our *part-of-speech tagger* (POS tagger), which marks each fragment in the method name with a certain *tag*. Informally, part-of-speech tagging means identifying the correct role of a word in a phrase or sentence.

Our POS tagger for method names is made simple, as the purpose is to provide a proof-of-concept, rather than create the optimal POS tagger for method names in Java. While there exist highly accurate POS taggers for regular English, their performance on Programmer English is unknown. Manual inspection of 500 tagged names taken from a variety of grammatical structures indicates that our POS tagger has an accuracy above 97%.

The POS tagger uses a primitive tag set: **verb**, **noun**, **adjective**, **adverb**, **pronoun**, **preposition**, **conjunction**, **article**, **number**, **type** and **unknown**. Examples of **unknown** fragments are misspellings ("anonomous"), idiosyncracies ("xget") and composites not handled by our decomposer ("nocando"). Less than 2% of the fragments in the corpus are **unknown**.

A fragment with a **type** tag has been identified as the name of a Java type. We consider a Java type to be in scope for the method name if the type is used in the method signature or body. This implies that the same method name can be interpreted differently depending on context. For instance, the method name `getInputStream` will be interpreted as the two fragments **get-InputStream** if `InputStream` is a Java type in scope of the method name, and as the three fragments **get-input-stream** otherwise. As illustrated by the example, we combine fragments to match composite type names. Type name ambiguity is not a problem for us, since we only need to know that a fragment refers to a type, not which one.



Fig. 2. Overview of the POS tagging process

The POS tagger operates in two steps, as shown in Fig. 2. First, we determine the range of *possible* tags for the fragments in the phrase, then we *select* a tag for each fragment. WordNet [7] is a core component of the first task. However, since WordNet only handles the four word classes verbs, nouns, adjectives and adverbs, we augment the results with a set of prepositions, pronouns, conjunctions and articles as well. Also, since Programmer English has many technical and specialized terms not found in regular English, we have built a dictionary of such terms. Examples include “encoder” and “optimizer” (nouns) and “blit” and “refactor” (verbs). The dictionary also contains expansions for many common abbreviations, such as “abbrev” for “abbreviation”.

The second step of the POS tagging is selection, which is necessary to resolve ambiguity. The tag selector is context-aware, in the sense that it takes into

account a fragment's position in a phrase, as well as the possible tags of surrounding fragments. For instance, the fragment **default** is tagged as **adjective** in the phrase **get-default-value**, and as **noun** in the phrase **get-default**. Since we know that method names tend to start with verbs, a fragment is somewhat more likely to be tagged **verb** if it is the first fragment in the phrase. Also, some unlikely interpretation alternatives are omitted because they are not common in programming. For instance, we ignore the possibility of **value** being a verb.

3.2 Semantic Analysis of Method Implementations

The goal of the semantic analysis of the method implementation is to derive a model of the method's behaviour. This model is an abstraction over the bytecode instructions in the implementation. In Frege's terms, we use the model to capture the programmer's idea of the method.

Attributes. In Sect. 2.4, we defined the semantics of a method m as a list of binary attribute values. The attributes are predicates, formally defined as conditions on Java bytecode. We have hand-crafted the attributes to capture various aspects of the implementation. In particular, we look at *control flow*, *data flow* and *state manipulation*, as well as the *method signature*. In addition, we have created certain attributes that we believe are significant, but that fall outside these categories.

The attributes are listed in Table 1. Each attribute is given a name and a short description. The formal definitions of the attributes range in sophistication, from checking for presence of certain bytecode instructions, to tracing the flow of parameter and field values.

Attribute Dependencies. In our previous work, we used strictly orthogonal attributes [3]. However, this sometimes forces us to choose between coarse and narrow attributes. As an example, we would have to choose between common, but not so distinguishing **Reads field** attribute, and the much more precise and semantically laden **Returns field value**. We therefore allow non-orthogonal attributes in our current work.

Table 2 lists the dependencies between the attributes. We see that they are straight-forward to understand. For instance, it should be obvious that all methods that return a field value must (a) read a field and (b) return a value.

Note that there are more subtle interactions at work between the attributes as well. For instance, **Throws exception** tends to imply **Creates objects**, since the exception object must be created at some point. However, it is not an absolute dependency, as rethrowing an exception does not mandate creating an object.

Critique. We have constructed the set of attributes under two constraints: our own knowledge of significant behaviour in Java methods and the relative simplicity of our program analysis. While we believe that the attributes are adequate for demonstration, we have no illusions that we have found the optimal set of attributes. A more sophisticated program analysis might allow us to define

Table 1. Attributes

<i>Control Flow</i>	
Contains loop	There is a control flow path that causes the same basic block to be entered more than once.
Contains branch	There is at least one jump or switch instruction in the bytecode.
Multiple return points	There is more than one return instruction in the bytecode.
Is recursive	The method calls itself recursively.
Same name call	The method calls a different method with the same name.
Throws exception	The bytecode contains an ATHROW instruction.
<i>Data Flow</i>	
Writes parameter value to field	A parameter value may be written to a field.
Returns field value	The value of a field may be used as the return value.
Returns parameter value	A parameter value may be used as the return value.
Local assignment	Use of local variables.
<i>State Manipulation</i>	
Reads field	The bytecode contains a GETFIELD or GETSTATIC instruction.
Writes field	The bytecode contains a PUTFIELD or PUTSTATIC instruction.
<i>Method Signature</i>	
Returns void	The method has no return value.
No parameters	The method has no parameters.
Is static	The method is static.
<i>Miscellaneous</i>	
Creates objects	The bytecode contains a NEW instruction.
Run-time type check	The bytecode contains a CHECKCAST or INSTANCEOF instruction.

Table 2. Attribute dependencies

Contains loop \Rightarrow Contains branch
Writes parameter value to field \Rightarrow Writes field \wedge \neg No parameters
Returns field value \Rightarrow \neg Returns void \wedge \neg Reads field
Returns parameter value \Rightarrow \neg Returns void \wedge \neg No parameters

or approximate interesting attributes such as **Pure function** (signifying that the method has no side-effects). It is also not clear that attributes are the best way to model the semantics of methods — for instance, the structure of implementations is largely ignored. However, the simplicity of attributes is also the strength of the approach, in that we are able to reduce the vast space of possible implementations to a small set of values that seem to capture their essence. This is important, as it facilitates comparing and contrasting the semantics of methods described by different phrases.

3.3 Phrase Semantics

The semantics of a single method captures the programmer’s subjective idea of what a method phrase means. When we gather many such ideas, we can approximate the sense of the phrase, that is, its objective meaning. We can group ideas by their concrete method phrases (names) such as **compare-to**, or by more abstract phrases containing tags, such as **find-[type]-by-[noun]**.

Phrase Characterization. Just like other natural language expressions, a phrase in Programmer English is only meaningful in contrast to other phrases. The English word *light* would be hard to grasp without the contrast of *dark*; similarly, we understand a phrase like **get-name** by virtue that it has different semantics from its opposite **set-name**, and also from all other phrases, most of which are semantically unrelated, such as **compare-to**. Since the semantics $\llbracket p \rrbracket$ of a phrase p is defined in terms of a list of attribute frequencies (see Sect. 2.4), we can characterize p simply by noting how its individual frequencies deviates from the average frequencies of the same kind.

For a given attribute a , the relative frequency $\xi_a(\mathcal{C}_n)$ for all names $n \in \mathcal{N}$ lies within the boundaries $0 \leq \xi_a(\mathcal{C}_n) \leq 1$. We divide this distribution into five named groups, as shown in Table 3. Each name is associated with a certain group for a , depending on the value for $\xi_a(n)$: the 5% of names with the lowest relative frequencies end up in the “low extreme” group, and so forth. This is a convenient way of mapping continuous values to discrete groups, greatly simplifying comparison.

Table 3. Percentile groups for attribute frequencies

$< 5\%$	Low extreme
$< 25\%$	Low
$25\% - 75\%$	Unlabelled
$> 75\%$	High
$> 95\%$	High extreme

Taken together, the group memberships for attributes a_i, \dots, a_k becomes an abstract characterization of a phrase, which can be used to generate a description of it.

3.4 Method Delegation

The use of method delegation in Java programs — invoking other methods instead of defining the behaviour locally — is a challenge for our analysis. The reason is that a method implementation that delegates directly to another method exposes no behaviour of its own, and so it “waters down” the semantics of the method name.

There are two simple ways of handling delegation: inlining and exclusion. Inlining essentially means copying the implementation of the called method into the implementation of the calling method. There are several problems with inlining. First, it undermines the abstraction barrier between methods and violates the encapsulation of behaviour. Second, it skews the analysis by causing the same implementation to be analyzed more than once. We therefore prefer exclusion, which means that delegating methods are omitted from the analysis. However, what constitutes delegation is fuzzy: should we ignore methods that calculate parameters passed to other methods, or methods that delegate to a sequence of methods? For simplicity, we only identify and omit single, direct delegation with an equal or reduced list of parameters.

4 Engineering the Phrase Book

This section describes the engineering efforts undertaken to produce the proof-of-concept phrase book for Java programmers. In particular, we describe how we meet the requirements outlined in Sect. 2.2, and take a closer look at the algorithm used to generate the phrase book.

4.1 Meeting the Requirements

In Sect. 2.2, we mandated that the phrase book be relevant and useful.

Relevance. We fulfill the relevance requirement by using a large corpus of Java applications as data for our analysis. This ensures that the results reflect actual real-world practice. The corpus is the same set of applications we used in our previous work [3]. Since many applications rely on third-party libraries, the corpus has been carefully pruned to ensure that each library is analyzed only once. This is important to avoid skewing of the results: otherwise, we cannot be certain that the semantics of a phrase reflects the cross-section of many different implementations.

The corpus consists of 100 open-source applications and libraries from a variety of domains: desktop applications, programmer tools, programming languages, language tools, middleware and frameworks, servers, software development kits, XML tools and various common utilities. Some well-known examples include the Eclipse integrated development environment, the Java software development toolkit, the Spring application framework, the JUnit testing framework, and the Azureus bittorrent client³. Combined, the applications in the corpus contain more than one million methods.

³ Azureus is currently the most downloaded and actively developed application from SourceForge.net.

Usefulness. In order to produce a phrase book that is as useful as possible, we want the phrase book to be short, easy to read, and containing only the most useful phrases. Here, we present our translation of the qualitative usefulness requirements into quantitative ones.

- *Validity.* Each phrase must represent at least 100 methods.
- *Precision.* Intuitively, precision means how consistently a phrase refers to the same semantics. Since entropy measures the independence of attributes, entropy is an inverse measurement of precision. Each phrase representing a refinement of another phrase must therefore lead to *decreased* entropy, corresponding to *increased* precision.
- *Ubiquity.* Each phrase must be present in at least half of the applications in the corpus.

Tweaking the actual numbers in these criteria allows us to control the size of the phrase book. The values we have chosen yields a phrase book containing 364 phrases. The ideal size of the phrase book is a matter of taste; we opt for a relatively small one compared to natural-language dictionaries.

4.2 Generation Algorithm

Below, we present and explain the pseudo-code (Fig. 3) for the algorithm that automatically generates the phrase book. Note that the pseudo-code glosses over many details to highlight the essentials of the algorithm. For brevity, we omit definitions for functions that are “self-explanatory”. The syntax is influenced by Python, meaning that indentation is significant and used to group blocks.

The pseudo-code outlines a fairly simple recursive algorithm. The driving function is `refine`, which generates a refinement of a phrase. Note that each phrase implicitly defines a corpus of methods, so that a refinement of a phrase also means a narrowing of the corpus.

```

refine(phrase):
    for tag in tags:
        t-phrase = phrase-append(phrase, tag)
        if useful(t-phrase, phrase):
            used-phrases = ()
            for f-phrase in fragment-phrases(p, tag):
                if useful(f-phrase, t-phrase):
                    used-phrases.add(f-phrase)
                    write-entry(f-phrase)
                    refine(f-phrase)
            r-phrase = mark-special(t-phrase)
            if useful(r-phrase, phrase):
                write-entry(r-phrase)
                refine(r-phrase)

```

Fig. 3. Pseudo-code for the phrase book generation algorithm

First, we iterate over the tags in our tag set (see Sect. 3.1). For each tag, we create a new phrase representing a refinement to only the methods whose names satisfy the new tag. We demand that this refinement be useful, or we ignore the entire tag. The refinement is useful if it meets the criteria of the **useful** function. This function embodies the criteria discussed in Sects. 2.2 and 4.1.

If the refinement is useful, we try to find even more useful refinements using fragments instead of the tag. Assume that we are calling **expand** on the phrase **get-***. We expand the phrase with the tag **noun**, yielding the new phrase **get-[noun]-***. Finding the new phrase to be useful, we generate more concrete refinements such as **get-name-*** and **get-customer-***. If they are useful, we call the **write-entry** function, which generates a description that is included in the phrase book, and recurse, by calling **refine** on the concrete refinement. Finally, we examine the properties of the corpus of remnant methods; those that match the tag phrase, but are not included in any of the useful concrete refinements. We say that these methods are captured by a special phrase **r-phrase**. The **r-phrase** is equal to the **t-phrase**, except that it potentially captures fewer method names, and hence might represent a smaller corpus. For instance, if **get-name-*** is useful and **get-customer-*** is not, then **r-phrase** captures the phrases captured by **get-[noun]-***, except those also captured by **get-name-***. If **r-phrase** is useful, it is included in the phrase book. Note that if no useful concrete refinements are found, **r-phrase** degenerates to **t-phrase**.

5 Results

While the phrase book has been designed for brevity, it is still much too large to be included in this paper. We therefore present some excerpts highlighting different aspects. The full version is available at <http://phrasebook.nr.no>. We also take a look at the distribution of grammatical structures, and using the phrase book to guide naming.

Terminology. Table 4 explains the basic terminology used in the phrase book. In addition, we use the modifier *comparatively* to indicate that the frequency is low despite being in the higher quantiles, or high despite being in the lower quantiles. For instance, a phrase might denote methods that call themselves recursively *more often* than average methods, even if the actual frequency might be as low as 0.1.

Example Entry. To illustrate how the data uncovered by our analysis is presented in the phrase book, we show the entry for the phrase **find-[type]**. It is an interesting example of a slightly abstract phrase with a clear meaning.

find-[type]. These methods very often contain loops, use local variables, have branches and have multiple return points, and often throw exceptions, do runtime type-checking or casting and are static. They rarely return void, write parameter values to fields or call themselves recursively.

Table 4. Phrase book terminology

<i>Phrase</i>	<i>Meaning</i>
Always	The attribute value is always 1.
Very often	Frequency in the high extreme percentile group.
Often	Frequency in the high percentile group.
Rarely	Frequency in the low percentile group.
Very rarely	Frequency in the low extreme percentile group.
Never	The attribute value is always 0.

Each entry in the phrase book describes what signifies the corpus of methods captured by the phrase; that is, how it differs from the average (Sect. 3.3). We find no surprises in the distinguishing features of **find-[type]**; in fact, it is a fairly accurate description of a typical implementation such as:

```
Person findPerson(String ssn) {
    Iterator itor = list.iterator();
    while (itor.hasNext()) {
        Person p = (Person) itor.next();
        if (p.getSSN().equals(ssn)) {
            return p;
        }
    }
    return null;
}
```

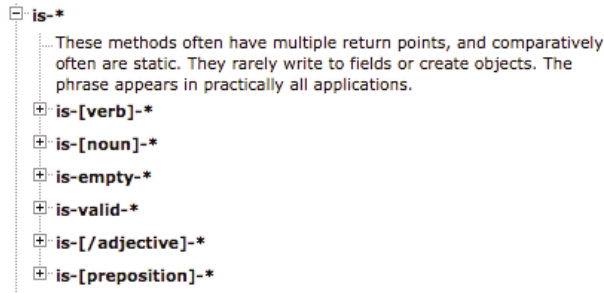
We iterate over a collection of objects (**Contains loop**), cast each object to its proper type (**Run-time type check**) and store it in a variable (**Local assignment**), and terminate early if we find it (**Multiple returns**). A common variation would be to throw an exception (**Throws exception**) instead of returning `null` if the object could not be found.

Refinement. As explained in Sect. 4.1, the phrase book is engineered to yield useful entries, understood as valid, precise and ubiquitous ones. The generation algorithm has been designed to prefer concrete phrases over abstract ones, as long as the criteria for usefulness are fulfilled.

One effect of this strategy is that the everyday “cliché” methods `equals`, `hashCode` and `toString` defined on `Object` are not abstracted: they emerge as the concrete phrases **equals**, **hash-code** and **to-String**. This is not surprising, as the names are fixed and the semantics are well understood.

The algorithm’s ability to strike the right balance between concrete and abstract phrases is further illustrated by the branch for the **is-*** phrase, shown in Fig. 4.

We see that the algorithm primarily generates abstract refinements for **is-***; one for each of the tags **verb**, **noun**, **adjective** and **preposition**. However, in the case of **adjective**, two concrete instances are highlighted: **is-empty-*** and

Fig. 4. The `is-*` branch of phrases

is-valid-*. This matches nicely with our intuition that these represent common method names. We write `[/adjective]` for the subsequent phrase to indicate that it captures adjectives except the preceding “empty” and “valid”.

Grammar. We find that the vast majority of method phrases have quite degenerate grammatical structures. By far the most common structure is `[verb]-[noun]`. Furthermore, compound nouns in Programmer English, as in regular English, are created by juxtaposing nouns. The situation becomes even more extreme when we collapse these nouns into one, and introduce the tag `noun+` to represent a compound noun. The ten most common grammatical structures are listed in Table 5.

Table 5. Distribution of grammatical structures

<i>Structure</i>	<i>Instances</i>	<i>Percent</i>
<code>[verb]-[noun+]</code>	422546	39.45%
<code>[verb]</code>	162050	15.13%
<code>[verb]-[type]</code>	78632	7.34%
<code>[verb]-[adjective]-[noun+]</code>	74277	6.93%
<code>[verb]-[adjective]</code>	28397	2.65%
<code>[noun+]</code>	26592	2.48%
<code>[verb]-[noun+]-[type]</code>	18118	1.69%
<code>[adjective]-[noun+]</code>	15907	1.48%
<code>[noun+]-[verb]</code>	14435	1.34%
<code>[preposition]-[type]</code>	13639	1.27%

Guidance. Perhaps the greatest promise of the phrase book is that it can be used as guidance when creating and naming new methods. Each description could be translated to a set of rules for a given phrase. An interactive tool, e.g., an Eclipse plug-in, could use these rules to give warnings when a developer breaches them.

As an example, consider the phrase **equals**, which the phrase book describes as follows:

equals. These methods very often have parameters, call other methods with the same name, do runtime type-checking or casting, have branches and have multiple return points, and often use local variables. They never return field values or return parameter values, and very rarely return void, write to fields, write parameter values to fields or call themselves recursively, and rarely create objects or throw exceptions. The phrase appears in most applications.

The extreme clauses are most interesting, because they represent the clearest characteristics for the phrase. For instance, no programmer contributing to the corpus has ever let an **equals** method return a value stored in a field — a strong suggestion that you might not want to do so either! However, we note that the phrase book reflects the actual use of phrases, not the ideal use. This means that the description might capture systematic implementation problems; i.e., common malpractice for a given phrase. We might look for clues in the negative clauses, indicating rare — even suspicious — behaviour. For instance, we see that there are **equals** methods that create objects and throw exceptions, which might be considered dubious. More severely, recursion in an **equals** method sounds like a possible bug. Indeed, inspection reveals an *infinite recursive loop* bug in an **equals** method in version 1.0 of Groovy⁴.

We conclude that the rules uncovered by the phrase book appear to be useful as input to a naming-assistance tool. However, the rules might need to be tightened somewhat, to compensate for fallible implementations in the corpus. After all, the corpus reflects the current state of affairs for naming, and the aim of a naming-assistance tool would be to improve it.

6 Related Work

We build on our previous work [3], which defined semantics for *action verbs*, the initial fragment of method names. We summarized the findings in *The Programmer's Lexicon*, an automatically generated description of the most common verbs in a large corpus of Java applications. The distinguishing characteristic of our work, is that we compare the names and semantics of methods in a large corpus of Java applications.

Other researchers have analyzed Java applications in order to describe typical Java programmer practice. Collberg et al. [8] present a large set of low-level usage statistics for a huge corpus of Java programs. Examples of statistics included are the number of subclasses per class, the most common method signatures and bytecode frequencies. Baxter et al. [9] have similar goals, in using statistics to describe the anatomy of real-world Java programs. In particular, they investigate the claim that many important relationships between software artifacts follow a “power-law” distribution. However, none of these statistics are linked to names.

There have also been various kinds of investigations into identifiers, traditionally in the context of program comprehension. Lawrie et al. [10] study how the quality of identifiers affect the understanding of source code. Caprile and

⁴ <http://groovy.codehaus.org/>

Tonella [11] investigate the structure of function identifiers as found in C programs. They build a dictionary of identifier fragments and propose a grammar for identifiers, but make no attempt at defining identifier semantics. Antoniol et al. [12] find that the names used in programming evolve more slowly than the structures they represent. They argue that the discrepancy is due to lack of tool support for name refactoring. In this work, names are linked to structures, but not the semantics of the structures.

Lately, more interest can be seen in investigating and exploiting name semantics. Singer et al. [13] share our ambition in ascribing semantics to names based on how they are used. They analyze a corpus of real-world Java applications, and find evidence of correlation between type name suffixes (nouns) and some of the micro patterns described by Gil and Maman [14]. Micro patterns are formal, traceable conditions defined on Java types.

Pollock et al. [15] investigate various ways of utilizing “natural language clues” to guide program analysis. Shepherd et al. [16] apply method name analysis to aid in aspect mining. In particular, they investigate the relationships between verbs (actions) and nouns (types) in programs. The scattering of the same verb throughout a program is taken as a hint of a possible cross-cutting concern. Finally, Ma et al. [17] use identifier fragments to index software repositories, to assist in querying for reusable components. These works involve exploiting implicit name semantics, in that relationships between names are taken to be meaningful. However, what the semantics are remains unknown. Our work is different, in that we want to explicitly model and describe the semantics of each name.

7 Conclusion

The names and implementations of methods are mutually dependent on each other. The phrase book contains descriptions that captures the objective *sense* of the phrases, that is, the common understanding among Java programmers of what the phrases mean. We arrived at the sense by correlating over a million method names and implementations in a large corpus of Java applications. By using attributes, defined as predicates on Java bytecode, we modeled the semantics of individual methods. By aggregating methods by the phrases that describe them, we derived the semantics of the phrases themselves. From the semantics, we generated the textual descriptions gathered in the phrase book.

We believe that further investigation into the relationship between names and implementations can yield more valuable insight and contribute to improved naming. Currently, we are refining our model of the semantics of methods, in order to make it more sophisticated and precise. This will allow us to more accurately describe the meaning of the phrases. An obvious enhancement is to use a state-of-the-art static analysis tool to provide a richer, more descriptive set of attributes. We are also considering developing a model for the semantics that better captures the structure of the implementations. At an abstract level, it might be possible to identify machine-traceable *patterns* for method

implementations. Inspired by Singer et al. [13], then, we might look for correlation between names and these patterns.

While we have shown that names do have grammatical structure, we believe that the potential for natural language expression in names is under-utilized. Indeed, by far the most common structure is the simple **[verb]-[noun]** structure. Longer, more complex names gives the possibility of much more precise descriptions of the behaviour of methods. Improved tool support for verifying name quality might motivate programmers to exploit this possibility to a greater extent than at the present.

We are working on transforming the results presented in this paper into a practical tool, supporting a much richer set of naming conventions than adherence to simple syntactic rules such as the camel case convention. The tool will warn against dissonance between name and implementation, and suggest two paths to resolution: 1) select a more appropriate name from a list proposed by the tool, or 2) perform one or more proposed changes to the implementation.

In a somewhat longer timeframe, the tool could be extended to support grammatical conventions as well. An example would be to warn against mixing verbose and succinct naming styles. One might debate whether it is better to explicitly mention types in method names (e.g., `Customer findCustomerByOrder(Order)`) or not (e.g., `Customer find(Order)`), but to mix both styles in the same application is definitely confusing. Tool support could help achieve grammatical consistency within the application.

References

1. Edwards, J.: Subtext: uncovering the simplicity of programming. In: [18], pp. 505–518
2. Frege, G.: On sense and reference. In: Geach, P., Black, M. (eds.) *Translations from the Philosophical Writings of Gottlob Frege*, pp. 56–78. Blackwell, Malden (1952)
3. Høst, E.W., Østvold, B.M.: The programmer's lexicon, volume I: The verbs. In: *SCAM 2007: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, Washington, DC, USA, pp. 193–202. IEEE Computer Society, Los Alamitos (2007)
4. Wittgenstein, L.: *Philosophical Investigations*. Prentice Hall, Englewood Cliffs (1973)
5. Cover, T.M., Thomas, J.A.: *Elements of Information Theory*, 2nd edn. Wiley Series in Telecommunications. Wiley, Chichester (2006)
6. Manning, C.D., Schuetze, H.: *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge (1999)
7. Fellbaum, C.: *WordNet: An Electronic Lexical Database*. MIT Press, Cambridge (1998)
8. Collberg, C., Myles, G., Stepp, M.: An empirical study of Java bytecode programs. *Software Practice and Experience* 37(6), 581–641 (2007)
9. Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., Tempero, E.: Understanding the shape of Java software. In: *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006*, Portland, Oregon, USA, pp. 397–412. ACM, New York (2006)

10. Lawrie, D., Morrell, C., Feild, H., Binkley, D.: What's in a name? A study of identifiers. In: Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006), Athens, Greece, pp. 3–12. IEEE Computer Society, Los Alamitos (2006)
11. Caprile, B., Tonella, P.: Nomen est omen: Analyzing the language of function identifiers. In: Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE 1999), Atlanta, Georgia, USA, pp. 112–122. IEEE Computer Society, Los Alamitos (1999)
12. Antonial, G., Guéhéneuc, Y.G., Merlo, E., Tonella, P.: Mining the lexicon used by programmers during software [sic] evolution. In: Proceedings of the International Conference on Software Maintenance (ICSM), pp. 14–23 (2007)
13. Singer, J., Kirkham, C.: Exploiting the correspondence between micro patterns and class names. In: SCAM 2008: Proceedings of the Eight IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 67–76. IEEE Computer Society, Los Alamitos (2008)
14. Gil, J., Maman, I.: Micro patterns in Java code. In: [18], pp. 97–116
15. Pollock, L.L., Vijay-Shanker, K., Shepherd, D., Hill, E., Fry, Z.P., Maloor, K.: Introducing natural language program analysis, pp. 15–16. ACM, New York (2007)
16. Shepherd, D., Pollock, L.L., Vijay-Shanker, K.: Towards supporting on-demand virtual remodularization using program graphs. In: Filman, R.E. (ed.) AOSD, pp. 3–14. ACM, New York (2006)
17. Ma, H., Amor, R., Tempero, E.D.: Indexing the Java API using source code. In: Australian Software Engineering Conference, pp. 451–460. IEEE Computer Society, Los Alamitos (2008)
18. OOPSLA 2005. Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16–20, 2005, San Diego, CA, USA. ACM, New York (2005)

Author Index

- Aßmann, Uwe 134
Alves, Tiago L. 285
- Basten, H.J.S. 265
Bravenboer, Martin 74
Brown, John 134
- de Lara, Juan 54
Demirezen, Zekai 178
Dingel, Juergen 245
Drivalos, Nikolaos 151
- Ekman, Torbjörn 95
- Fernandes, Kiran J. 151
Fritzsche, Mathias 134
Fuentes, Lidia 188
- Garcia, Alessandro 188
Gilani, Wasif 134
Gjøsæter, Terje 305
Goldschmidt, Thomas 168
Gray, Jeff 178
Guerra, Esther 54
- Hage, Jurriaan 35
Hedin, Görel 95
Høst, Einar W. 322
- Isfeldt, Ingelin F. 305
- Janssens, Dirk 208
Johannes, Jendrik 134
Johnson, Ralph E. 114
Jouault, Frédéric 178
- Keller, Anne 208
Kilpatrick, Peter 134
Kleppe, Anneke 1
Klint, P. 265
Kolovos, Dimitrios S. 151
- Liang, Hongzhi 245
Loughran, Neil 188
- Mitschke, Simon 134
Moody, Daniel 16
- Nilsson-Nyman, Emma 95
- Østvold, Bjarte M. 322
Overbey, Jeffrey L. 114
- Paige, Richard F. 151
Prinz, Andreas 305
- Rivera, José Eduardo 54
- Sánchez, Pablo 188
Schätz, Bernhard 227
Spence, Ivor 134
Sun, Yu 178
- Tairas, Robert 178
- Vallecillo, Antonio 54
van den Brand, M.G.J. 8
Van Gorp, Pieter 208
van Hillegersberg, Jos 16
van Keeken, Peter 35
Visser, Eelco 74
Visser, Joost 285