Lecture Notes in Computer Science 2426

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Jean-Michel Bruel Zohra Bellahsene (Eds.)

Advances in Object-Oriented Information Systems

OOIS 2002 Workshops
Montpellier, France, September 2, 2002
Proceedings

Springer

Volume Editors

Jean-Michel Bruel
LIUPPA, Computer Science Research Department
University of Pau
B.P. 1155, 64013 Pau Cedex, France
E-mail: Jean-Michel.Bruel@univ-pau.fr

Zohra Bellahsene
LIRMM
161 rue Ada, 34392 Montpellier Cedex 5, France
E-mail: bella@lirmm.fr

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Advances in object oriented information systems : OOIS 2002 workshops, Montpellier, France, September 2, 2002 ; proceedings / Jean-Michel Bruel ; Johra Bellahsene (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2002
(Lecture notes in computer science ; Vol. 2426)
ISBN 3-540-44088-7

CR Subject Classification (1998): H.2, H.4, H.5, H.3, I.2, D.2, D.4, K.4.4, J.1

ISSN 0302-9743
ISBN 3-540-44088-7 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by Da-TeX Gerd Blumenstein
Printed on acid-free paper SPIN: 10873861 06/3142 5 4 3 2 1 0

Preface

For the first time four workshops have been held in conjunction with the 8th Object-Oriented Information Systems conference, OOIS 2002, to encourage interaction between researchers and practitioners. Workshop topics are, of course, inline with the conference's scientific scope and provide a forum for groups of researchers and practitioners to meet together more closely and to exchange opinions and advanced ideas, and to share preliminary results on focused issues in an atmosphere that fosters interaction and problem solving.

The conference hosted four one-day workshops. The four selected workshops were fully in the spirit of a workshop session hosted by a main conference. Indeed, OOIS deals with all the topics related to the use of object-oriented techniques for the development of information systems. The four workshops are very specific and contribute to enlarging the spectrum of the more general topics treated in the main conference. The first workshop focused on a very specific and key concept of object-oriented development, the specialization/generalization hierarchy. The second one explored the use of "non-traditional" approaches (at the edge of object-oriented techniques, such as aspects, AI, etc.) to improve reuse. The third workshop dealt with optimization in Web-based information systems. And finally the fourth workshop investigated issues related to model-driven software development.

Each workshop was organized by a group of international organizers, leading a program committee in the process of reviewing submissions. Together the workshops selected 30 papers, involving about 80 authors, and gathered a good number of participants to the campus of the University of Montpellier on September 2, 2002.

The editors would like to thank Springer-Verlag for publishing this year both the main conference and workshops proceedings in the *Lecture Notes in Computer Science* series. They would also like to thank all the workshop organizers and program committee members for their support and collaboration in the success of this first series of workshops and in the preparation of this volume. Finally, they are also grateful to the local organizers for their support.

September 2002

Jean-Michel Bruel
Zohra Bellahsene

Organization

This volume is a compilation of the four OOIS workshops organized at the University of Montpellier. It is organized in four chapters. Each chapter contains an introduction, written by the workshop organizers, which provides an overview of the workshop contribution, along with the Program Committee and details of other related information, followed by the accepted papers of the workshop.

The proceedings of the main conference were published as LNCS Vol. 2425.

OOIS 2002 Executive Committee

General Chair:	Colette Roland, Paris I University, France
Program Co-Chairs:	Zohra Bellahsène, LIRMM, France Dilip Patel, South Bank University, UK
Workshops:	Jean-Michel Bruel, LIUPPA, France Computer Science Research Department, B.P. 1155, F-64013, Pau Université, Cedex, France E-mail: Jean-Michel.Bruel@univ-pau.fr

Table of Contents

MAnaging SPEcialization/Generalization HIerarchies

Reuse in OO Information Systems Design

Efficient Web-Based Information Systems

Model-Driven Approaches to Software Development

MAnaging SPEcialization/Generalization HIerarchies

Marianne Huchard[1], Hernan Astudillo[2], and Petko Valtchev[3]

[1] L.I.R.M.M., France
[2] Financial Systems Architects, New York, USA
[3] Université de Montréal, Canada

Preface

In object-oriented approaches (modeling, programming, databases, knowledge representation), the core of systems is, most of the time, a specialization hierarchy, that organizes concepts of the application domain or software artifacts useful in the development. These concepts are usually known as classes, interfaces and types. Software Engineering methods for design and analysis are concerned by application domain modeling as well as transferring the model into the target programming language chosen for implementation. For programming languages and database systems, the specialization hierarchy is implemented by inheritance, that also supports feature (specification or code) sharing and reuse. In Knowledge Representation and data-mining approaches, the modeling aspect of a class hierarchy prevails, whereas its main purpose is to guide the process of reasoning and rule discovery.

Despite their wide and long use in these domains, specialization hierarchies still give rise to controversial interpretations and implementations. The design, implementation and maintenance of such hierarchies are complicated by their size, the numerous and conflicting generalization criteria, and the natural evolution of the domains themselves and of the knowledge about them, which of course must be reflected by the hierarchies.

The fact that two workshops ("The Inheritance Workshop" at ECOOP 2002 [2] and *MASPEGHI* at OOIS 2002) hold the same year on close topics indicates that it is time to bring to the fore specialization/generalization as a specific research field.

Among the 15 early submissions, we selected 9 papers that cover five main areas:

- general discussion about common sense specialization and its implementation,
- lattice/order theory (aspects useful for hierarchy manipulation),
- modeling (points of view and new paradigms),
- programming (analysis of practices, meta-programming),
- linguistic issues (taxonomy construction).

The web site of *MASPEGHI* remained open until the workshop for refereed late submissions that are gathered in [1].

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 1–2, 2002.

Organization

The workshop was organized by Marianne Huchard (LIRMM, France), Hernan Astudillo (Financial Software Architects, USA) and Petko Valtchev (Université de Montréal, Canada).

Program Committee

Michel Dao (France Télécom R&D, France)
Robert Godin (UQAM, Canada)
Haim Kilov (Financial Systems Architects, USA)
Thérèse Libourel (LIRMM, France)
Juan Lloréns (Universidad Carlos III, Spain)
Joaquin Miller (Financial Systems Architects, USA)
Amedeo Napoli (INRIA Lorraine, France)
Ruben Prieto-Diaz (James Madison University, USA)
Derek Rayside (University of Toronto, Canada)
Houari Sahraoui (Université de Montréal, Canada)
Markku Sakkinen (Jyväskylän yliopisto, Finland)
Gregor Snelting (Universität Passau, Germany)

Additional Referees

Y. Ahronovitz, G. Ardourel, R. Ducournau, M. Lafourcade, C. Roume (LIRMM, France); Gillian Gass, Sara Scharf (University of Toronto, Canada).

Primary Contact

For more details about the workshop please contact:
Marianne Huchard
LIRMM, CNRS et Université Montpellier 2
161, rue Ada – 34392 Montpellier cedex 5, France
email: `huchard@lirmm.fr`
tel: +33 (0)4 67 41 86 58
fax: +33 (0)4 67 41 85 00
url: `http://www.lirmm.fr/~huchard/MASPEGHI/`

References

1. M. Huchard, H. Astudillo and P. Valtchev (editors) *Late Submissions of the workshop Managing SPEcialization/Generalization HIerarchies (MASPEGHI), OOIS'2002* Research Report LIRMM, CNRS et Université Montpellier 2, n.02087, August 2002. 1
2. A. Black, E. Ernst, P. Grogono and M. Sakkinen (editors) *Proceedings of the Inheritance workshop at ECOOP 2002* Publications of Information Technology Research Institute, University of Jyväskylä, 12/2002, ISBN: 951-39-1252-3. 1

"Real World" as an Argument for Covariant Specialization in Programming and Modeling

Roland Ducournau

L.I.R.M.M., Université Montpellier 2
161, rue Ada – 34392 Montpellier cedex 5, France
ducournau@lirmm.fr
http://www.lirmm.fr/~ducour/

Abstract. Class specialization is undoubtedly one of the most original and powerful features of object orientation as it structures object models at all stages of software development. Unfortunately, the semantics of specialization is not defined with the same accuracy in the various fields. In programming languages, specialization is constrained by type theory and by a type safe policy, whereas its common sense semantics dates back to the Aristotelian tradition. The well known covariant vs. contravariant controversy originates here. In this paper, we investigate how modeling and programming languages deal with this mismatch. We claim that type errors are part of the real world, so they should be taken into account at all stages of software development. Modeling as well as programming languages should adopt a covariant policy.

1 Introduction

Originated in Simula more than 30 years ago [3], object orientation has become, by now, quite hegemonic in the field of programming languages and software engineering, not to speak of databases or knowledge representation. This hegemony has often been explained by the closeness of various object-oriented concepts to corresponding common sense notions as they have been elaborated in classic philosophy [21,22]. Noticing that, one could hope for a *seamless* development process from so-called real world to program implementation, through analysis and design steps. However, this apparently uniform model presents some discontinuities, particularly when specialization is concerned.

Class specialization is undoubtedly one of the most original and powerful features of object orientation, yielding most of its qualities and breaking with previous programming paradigms. A large part of the literature is devoted to it, and it is the central point of many active topics of research such as inheritance (programming languages), classification or subsumption (knowledge representation), polymorphism or subtyping (type theory). Unfortunately, the semantics of specialization is not defined with the same accuracy in those various fields. Moreover, specialization may be constrained, in some field, by some external considerations. For instance, the well known *covariant vs. contravariant controversy* (e.g. [8], [18, chapter 17] or [25]) can be explained as a conflict between the

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 3–13, 2002.

demands of a type safe policy and the needs for expressivity. In this paper, we look at this well known controversy from the point of view of our common sense understanding of the "real world" and investigate whether modeling languages answer adequately to this requirement. Type errors are part of the real world. A dramatic example has been given by the "mad cow" disease: `cows`, as a specialization of `herbivorous`, should only eat `grass`, not `meat`, but it happened that they were feeded with remains of cows. So, we claim that type errors should be taken into account at all stages of software development: analysis and design methods, as well as programming languages should adopt a covariant policy.

The rest of this paper is organized as follows: section 2 briefly recalls the *de facto* standard object model, then states how specialization can be related to common sense reasoning and Aristotelian tradition and gives some hints regarding how knowledge representation formalizes it. Next section takes the viewpoint of programming languages and type theory and states the covariance vs. contravariance controversy. The case of most widely used languages is examined and some alternatives such as multiple dispatch are investigated. Section 4 looks at analysis and design methods, mainly UML, and concludes to their current abdication to impose a semantics in front of JAVA's one. In conclusion, we sketch out the specifications of a language adapted to the semantics of specialization.

2 Semantics of Specialization

The *de facto* standard object model is the class-based model, consisting of *classes*, organized in a *specialization* hierarchy, and *objects* created as instances of those classes by an instantiation process. Each class is described by a set of properties, *attributes* for the state of its instances and *methods* for their behavior. Applying a method to an object follows the metaphor of *message sending* (also called *late binding*): the invoked method is selected according to the class of the object (called the *receiver*). This is the core of the model and it suffices to state the point of the specialization semantics. It is a *de facto* standard since it covers all of the widely used languages as the core of analysis and design models.

Though novel in computer science, specialization has quite ancient roots in the Aristotelian tradition, in the well known syllogism: *Socrates is a human, humans are mortals, thus Socrates is a mortal*. Here *Socrates* is an instance, *human* and *mortal* are classes. The interested reader will find in [21,22] a deep analysis of the relationships between object orientation and Aristotle syllogistic.

2.1 Inclusion of Extensions, Intensions and Domains

According to the Aristotelian tradition, as revised with the computer science vocabulary, one can generalize this example by saying that *instances of a class are also instances of its superclasses*. More formally, $\prec$ is the specialization relationship ($B \prec A$ means that B is a subclass of A) and Ext is a function which maps classes to the sets of their instances, their *extensions*. Then:

$$B \prec A \Longrightarrow Ext(B) \subseteq Ext(A) \tag{1}$$

This is the essence of specialization and it has two logical consequences: inclusion of intensions (i.e. inheritance) and inclusion of properties' domains (i.e. covariant refinement). When considering the properties of a class, one must remember that they are properties of instances of the class, factorized in the class. Let B be a subclass of A: instances of B being instances of A, have all the properties of instances of A. One says that subclasses inherit properties from superclasses. More formally, Int is a function which maps classes to the sets of their properties, their *intensions*:

$$B \prec A \Longrightarrow Int(A) \subseteq Int(B) \tag{2}$$

Properties have a value in each object and can be described in the class by a *domain*, that is the set of values taken by the property in all the class's instances. For instance, the class `Person` has a property `age` whose domain is $[0, 120]$. When specializing a class, one refines the domains of inherited properties: for instance, a subclass `Child` of `Person` will have domain $[0, 12]$ for its property `age`. The function Dom maps classes and properties to sets of values. Then:

$$B \prec A \ \ \& \ \ P \in Int(A) \Longrightarrow Dom(B, p) \subseteq Dom(A, p) \tag{3}$$

The `age` example concerns attributes. Methods may have several domains, for parameters and returned value. As an example, consider classes of `Animals`, in a hierarchy à la Linnaeus, with a method `eat` defined with different domains in classes such as `herbivorous`, `carnivorous`, and so on. [18, chapitre 17] develops a longer example, more oriented towards programming languages.

The inclusions of extensions and intensions have opposite directions, while those of extensions and domains have the same: intensions can be said *contravariant* whereas domains are *covariant*, both w.r.t. extensions, i.e. specialization.

2.2 Specialization in Knowledge Representation

Though quite intuitive, inclusion (3) cannot be proved to be entailed by (1) without a careful definition of class extensions which needs a model-theoretic approach. Such a semantics of specialization has been formalized in knowledge representation systems called *description logics* or languages of the Kl-One family [27,10]. In previous works, we showed that such a formalization could be exported to a more standard object model but this is not a common approach [12]. A main feature of this semantics is that the equations corresponding to (1-3) can be equivalences, not mere implications: in other words, classes can be defined as necessary and sufficient conditions and specialization between classes (then called *subsumption*) can be deduced from class properties, which leads to *classification*. Previous examples obviously need such semantics since `adult` and `child` are defined by their `age`, as well as `herbivorous` and `carnivorous` by what they `eat`. However, such a semantics is not necessarily adapted to programming languages nor to analysis and design modeling, as it has a major drawback, being essentially monotonous: one can add values, not modify them. Nevertheless, it could give some hints to precise the semantics of object models, as well as semantical bases to automatic computation of class hierarchies [13].

3 Programming Languages, Subtyping and Polymorphism

Object-oriented programming languages can be considered as a mixture of object-oriented notions and programming languages notions. We will just consider the notion of type, central in programming languages, and focus on *statically typed* languages. Arguments in favor of static typing are numerous. The main one concerns reliability. Static, i.e. compile-time, analysis is needed to avoid dynamic, i.e. run-time, errors. Static typing allows a simple and efficient static analysis, whereas dynamic typing requires more expensive and less effective analyses. Anyway, static typing is another *de facto* standard.

3.1 Contravariance of Subtyping

In a statically typed language, every entity in the program text which can be bound to a value at run-time is annotated by a type, its *static type*. At run-time, every value has a type, its *dynamic type*, i.e. the class which creates the value as its instance. In such a context, an entity is said to be *polymorphic* when it can be bound to values of distinct types, and the dynamic types of the values must *conform to* the static type of the entity. Otherwise, there is a run-time type error, which may lead to an `unknown message` error when a method, called upon this entity, is known by the static type, not by the dynamic one.

Types and classes are quite similar—a type can be seen as a set of values (extension) and a set of operators (intension)—and the conformance relationship between types, denoted by ¡:, is analogous to specialization between classes. Statically typed languages allow a static (compile-time) type error checking, i.e. a *type safe* compilation. A simple way to allow this is to define conformance through the notion of *substitutability*: a type t_2 conforms to a type t_1 iff any expression of type t_1 can be substituted by (bound to) any value of type t_2 without any run-time type error. Types can be identified with classes or, preferably, types can be associated to classes but the very point is to liken class specialization and subtyping. Class specialization can support polymorphism—an instance of a subclass can be substituted to an instance of a superclass—as long as the type of the subclass conforms to the type of the superclass. Of course, with a type safe policy. Class specialization is thus constrained by type safety.

This constraint revolves around the way types of properties can be redefined (overridden) in a subclass. Let A be a class and m a method defined in A, noted m_A. Method types are noted in a functional way, with arrow types: m_A has, for instance, type $t \to t'$. Let B a subclass of A, where m is redefined in m_B, with type $u \to u'$. The type of B conforms to the type of A, only if $u \to u'$ is a subtype of $t \to t'$. Subtyping on arrow types is defined as follows [7]:

$$u \to u' \text{ ¡: } t \to t' \iff t \text{ ¡: } u \ \ \& \ \ u' \text{ ¡: } t' \tag{4}$$

A function of type $t \to t'$ can be replaced by a function of type $u \to u'$ if the latter accepts more values as parameter (t ¡: u) and returns less values (u' ¡: t'). Following Cardelli, the return type is said *covariant*, while the parameter type is *contravariant*: this is known as the *contravariance rule*. Attribute redefinition is

ruled by a mixture of them, as an attribute can be seen as a pair of two methods, a reader which returns the attribute value and a writer which set this value from its parameter's value. Thus an attribute redefinition must be both covariant and contravariant, leading to *invariance*.

3.2 Covariance vs. Contravariance

Coming back to the semantics of specialization, one sees that *domain* specialization is subjected to some kind of *covariant rule* (3). The controversy lies there: *contravariance* for types versus *covariance* for domains. The fact is that domains are not types. Domains are defined as the sets of values taken by a property—i.e. an attribute, a method parameter or returned value—on all instances of a class. Static types are program annotations intended to avoid run-time type errors. Domains are ruled by existential quantifiers—there exists a value for a property—, while types are ruled by universal quantifiers—any value of the type should be substitutable. Properties may have both domains and types. Domains can be understood as subsets of type extensions [6] and there is no way to precisely express domains in programming languages, but types. Nevertheless, introducing domains in programming languages would lead to domain errors and to a *domain safe* policy, which would be essentially the same as the type safe policy. So, our thesis is that domains can be expressed by types and that domain errors are part of the real world: cows should not eat meat, but grass, and there is no way to statically check for it. One unfortunately knows that, in the real world, it has not been dynamically checked. Type safe programs are certainly more reliable, but faithful programs are better. Thus the type safe policy should be imposed only when type errors originate in the program not when they are part of the real world.

3.3 Actual Languages

Actual languages are apparently ruled by the type safety dogma. C++ and Java, the two most widely used object-oriented languages, apply the contravariance rule, in a stricter way than needed: parameter and attribute types are actually invariant. As for return types, they are also invariant in Java, without any reason, but they can be covariantly redefined in C++, at least recently [15]. However, both languages present two error-proning features which make type safety unreachable. *Downcast* is a way to assume that a value of a given static type has actually a more specific subtype: this assumption must be checked at run-time, which leads to run-time type errors. *Static overloading* allows to define, in the same classes, different methods with the same name and distinct parameter types. Both features allow apparent covariant redefinition. In the case of static overloading, it remains an illusion since static overloading is ruled by a static dispatch which obviously cannot emulate dynamic dispatch but interferes with it in a very confusing way. As for downcast, it allows to precisely express the covariant semantics that a programmer would have to, at the cost of clumsiness and potential type errors. Moreover, downcast is not restricted to handling covariant parameters and can be reused as a bad general programming style.

EIFFEL is the only widely known language to rule out the contravariance rule: parameter and attribute types must be covariantly redefined [17,18]. However, it tries to maintain the type safety dogma, with the so-called *catcall* rule, where 'cat' stands for "Changing Availability or Type". We will state the rule in the case of a type change, i.e. a covariant refinement. A *call* is *polymorphic* if the receiver's dynamic type may be different from its static type. A *catcall* is a call to a method which is covariantly redefined in the subclasses of the receiver's type. *Polymorphic catcalls are forbidden.* Unfortunately, this rule would forbid to actually use the covariantly redefined methods, if it were applicable. For the *catcall* rule cannot be implemented in separate compilation and doesn't seem to have ever been implemented. Moreover, since a global analysis is needed, the type safety could be obtained with a far less strict rule [23].

3.4 Variations around Language Design

Many variations around standard object model and type system have been proposed, often as an answer to this controversy. We will examine two of them: variations on method dispatch and variations on polymorphism and subtyping.

Multiple Dispatch Multiple dispatch has been popularized by CLOS [4] before finding a theoretical framework in static typing [19,8]. It has been adopted by many languages. Apart from CLOS, which is dynamically typed, none of those languages is widely used: thus multiple dispatch remains academic, despite its true interest. In standard object model, method dispatch (also called late binding) is realized according to the message sending metaphor: the type of a distinguished parameter, called the receiver, is used to select a method. Other parameters have no influence on dynamic dispatch: they are only used for static overloading, when it is the case. With multiple dispatch, all parameters are used for selecting the method. An easy way to understand multiple dispatch is to see it as a single dispatch on the cartesian product of parameter types. This is the implicit CLOS point of view. Contravariance disappears as it concerns only parameters unused for dispatch but usual modularity of classes and methods also disappears: methods are no more within classes, but between.

Another view on multiple dispatch is provided by *overloaded functions*—not to confuse with static overloading—which preserve modularity [8]. *Multimethods*, instead of methods, are associated to classes: each multimethod can have several branches, i.e. methods, which differ from each other by the types of their secondary parameters. Dispatch is then two-steps: a multimethod single dispatch on the receiver type is followed by a branch multiple dispatch on the types of other parameters (i.e. single dispatch on their cartesian product). Some typing rules allow type safe compilation. Implementing covariant refinement of parameter types is easy with multimethods. One defines a multimethod with two branches: the first one is the overriding method, with refined parameter types, while the other one has non-refined parameter types and signals a type error.

Multiple dispatch is a good programming solution, for methods only since it doesn't apply to attributes. But it is no more type safe than, either downcast in C++ and JAVA, or a truly covariant language like EIFFEL. [5] proposes a

technique for automatically transform covariant programs into type safe multiple dispatch programs, by changing methods into multimethods and adding branches when meeting covariant refinement. Added branches call overridden multimethods. This technique may be adapted for pure covariant refinement: then, added branches signal a type error instead of calling overridden multimethod.

Genericity and Subtyping Many types systems have been proposed, often with the aim of making EIFFEL's type system safe. Besides its covariant policy, EIFFEL's type system presents an original feature. A type can be "anchored" to the type of the receiver (`self`, `this` or `current` according to languages) with the type `like current`, also called `mytype`. The anchor can also be a property p of `self`, with the type `like` p. Anchored types are typically covariant: in the interface of a class, i.e. on a non `self` receiver, they are not type safe, unless when used as return types. Several propositions for making EIFFEL type safe have been made [9,26]: they are mainly based on a translation of anchored types into parametric types. Indeed, as a corollary of the contravariance rule, if $A\langle T\rangle$ is a parametric type, B and C two classes, then $A\langle C\rangle$ is not a subtype of $A\langle B\rangle$ when $C \text{¡:} B$, at least in a type safe policy. Thus, those propositions for making EIFFEL type safe replace an unsafe covariant specialization by a safe but non substitutable parameterization.

Another way to avoid the contravariance controversy has been to dissociate subclassing and subtyping, as in SATHER [25]. Again, the result is to allow covariant refinement, but to forbid substitutability, as with the catcall rule or EIFFEL corrections. Not surprisingly, all those approaches are more or less equivalent, since usual types A need to be defined as *recursive types* $\mu(t)A(t)$ where μ binds a variable t to the type being defined, i.e. `mytype` [1]. Though genericity (i.e. parametric polymorphism) and subtyping (i.e. inclusion polymorphism) are conceptually different, their formal bases are the same. Thus any translation of anchored types into parametric types, or any dissociation of subclassing and subtyping will have no effect on the controversy: covariant refinement, polymorphism (i.e. substitutability) and type safety are incompatible.

4 Analysis and Design Methods

Analysis and design methods bridge the gap between, on the one hand, real world and common sense reasoning and, on the other hand, programming languages. Clearly, analysis should be independent from programming languages. There is no evidence that analysis' specificities and goals would be very different from knowledge representation's ones. In fact, one can find bridges between them (e.g. [14]) and the main requirement for knowledge representation is that it must afford a formal support to reasoning, while analysis should only produce an informal model. So, our first point will be that analysis methods should use covariant models, as common sense reasoning and knowledge representation.

As for design methods, it is unclear how independent from programming languages they should be. Part of design methods is often dedicated to implementation, thus depends on some specific languages.

Looking at existing methods, one finds a family of universal object-oriented methods, such as OMT [24], which have been unified in UML [20]. It doesn't matter that UML is not a method but a modeling language, since we are only concerned here by the object models. At the opposite, a language like EIFFEL [17] can be seen as equipped with its own design method [18]. Not surprisingly, this method preconizes a covariant modeling, thus satisfying our demands. UML being the product of a unification process, it is presumed to have gained experience from previous methods such as OMT. Moreover, it tends towards hegemony. So it will be our main target. UML, after OMT, demands that signatures, including return types, be invariant by overriding. Moreover, in OMT, static overloading is explicitly advocated. Though one of the goals of UML is to "*support specifications that are independent of particular programming languages*", it appears than many details of its specifications come from JAVA, such as signature invariance. A main drawback of the unifying approach of UML is then that unification applies only at lexical (i.e. entities composing a model) and syntactic (i.e. relationships between entities) levels: semantics is left to the reader or even is explicitly referred to the target programming language. In this article, we developed the case of the covariant semantics of specialization. Other features of modeling and programming languages could and should be discussed from the same point of view. For instance, the notion of visibility is realized in UML through a mixture of JAVA and C++ keywords, with a semantics which seems to be that of the programming languages, but it is well known that common keywords have different semantics in JAVA and C++ [2]. No effort seems to be made to propose a novel, proper semantics and the canonical encapsulation of SMALLTALK as well as the more complex export clauses of EIFFEL are not considered whereas they are closer to the essence of object orientation, by allowing the protection of `self`.

5 Conclusion and Perspectives

A main quality of object-oriented technology is to provide to programs, through various development stages, a uniform model from common sense understanding of "real world". In computer science and technology, this is an original and priceless quality which should lead to a *seamless* development process. One could expect that object-oriented languages reflect a conflict between formal theories which do impose some technical policy, e.g. type theory for type safety, and analysis methods, which would demand some expressivity, e.g. covariant refinement. Surprisingly, there is no evidence of such a conflict. Moreover, if a controversy does exist about co and contravariance, it is confined to the programming languages community. The UML community seems to find JAVA a perfect language and doesn't seem to have ever heard about EIFFEL, not to speak of Aristotle.

Nevertheless, analysis and design methods should see themselves as programming languages' clients: they should impose their specifications of what should be a programming language, instead of adopting the specifications of the current most widely used programming languages. This is not to rule out type theory: type safety is an important viewpoint on programs, not the main one.

Type errors exist in the "real world"—it could happen that cows eat meat, not grass, moreover it happened—thus, they should be integrated in analysis, design and programming stages. Programming languages designers must compromise between the languages expressivity and the formal properties they want to ensure. Such a compromise recalls the compromise between expressivity and completeness which caused a long lasting debate in the knowledge representation community [16]. But this new compromise differs for type safety is truly incompatible with the ability to express real world type errors.

Thus, our conclusion and perspectives are twofold. First, analysis and design methods, i.e. OMG, should adopt a model with covariant refinement integrating type errors both in the model and in the methods; and they should ask for programming languages adapted to such a model. Second, programming languages designers should design languages allowing an easy expression of covariant refinement, either with single or multiple dispatch; type errors should be explicitly handled at run-time, and an improved static analysis of those type errors should allow the programmer to compare the potential errors in the program with their existence in the analysis and design models. Analysts, designers and programmers should all be aware of the type errors which might happen when using covariant refinement. Moreover, if the ability to express covariance is necessary, invariance may often be sufficient: type safe models and programs must be encouraged. Languages could supply keywords to express that a specific property could or couldn't be covariantly refined in subclasses. As a corollary, there is no solution to the controversy to expect from type theory, for two reasons. First, type errors come from the "real world": no theory is powerful enough to modify it. Second, all the variations on types and polymorphism are mainly equivalent: a choice must always be made between specialization, polymorphism and type safety. The interested reader will find a more detailed discussion in [11]. More generally, for all features which are a matter of modeling, analysis and design methods should propose their own proper specifications to programming languages.

References

1. M. Abadi and L. Cardelli. On subtyping and matching. In W. Olthoff, editor, *Proc. ECOOP'95*, LNCS 952, pages 145–167. Springer-Verlag, 1995. 9
2. G. Ardourel and M. Huchard. Access graphs, another view on static access control for a better understanding and use. *J. of Object Technology*, 2002. (to appear). 10
3. G. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA Begin*. Petrocelli Charter, New York (NY), USA, 1973. 3
4. D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common Lisp Object System specification,. *ACM SIGPLAN Notices*, 23, 1988. 8
5. J. Boyland and G. Castagna. Type-safe compilation of covariant specialization: a practical case. In P. Cointe, editor, *Proc. ECOOP'96*, LNCS 1098, pages 3–25. Springer-Verlag, 1996. 8

6. C. Capponi, J. Euzenat, and J. Gensel. Objects, types and constraints as classification schemes. In G. Ellis, R. Levinson, A. Fall, and V. Dahl, editors, *Int. Conf. on Knowledge Re-use, Storage and Efficiency (KRUSE'95)*, pages 69–73, 1995. 7
7. L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. McQueen, and G. Plotkin, editors, *Semantics of Data Types*, LNCS 173, pages 51–67. Springer-Verlag, Berlin, 1984. 6
8. G. Castagna. *Object-oriented programming: a unified foundation*. Progress in Theoretical Computer Science Series. Birkhaüser, 1997. 3, 8
9. W. R. Cook. A proposal for making Eiffel type-safe. In S. Cook, editor, *Proc. ECOOP'89*, pages 57–70. Cambridge University Press, 1989. 9
10. F.-M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 191–236. CSLI Publications, Stanford (CA), USA, 1996. 5
11. R. Ducournau. "Real World" as an argument for covariant specialization in programming and modeling. RR 02-083, L. I. R. M. M., Montpellier, France, 2002. 11
12. R. Ducournau and G. Pavillet. Langage à objets et logique de descriptions : un schéma d'intégration. In I. Borne and R. Godin, editors, *Actes LMO'2001* in *L'Objet vol. 7*, pages 233–249. Hermès, 2001. 5
13. R. Godin, H. Mili, G. Mineau, R. Missaoui, A. Arfi, and T. Chau. Design of Class Hierarchies Based on Concept (Galois) Lattices. *Theory and Practice of Object Systems*, 4(2), 1998. 5
14. S. Greenspan, J. Mylopoulos, and A. Borgida. On formal requirements modeling languages: RML revisited. In *Int. Conf. on Software Engineering (ICSE'94)*, 1994. 9
15. A. Koenig. Standard – the C++ language. Report ISO/IEC 14882:1998, Information Technology Council (NCTIS), 1998. `http://www.nctis.org/cplusplus.htm`. 7
16. H. Levesque and R. Brachman. Expressiveness and Tractability in Knowledge Representation and Reasoning. *Computational Intelligence*, 3(2):78–93, 1987. 11
17. B. Meyer. *Eiffel: The Language*. Prentice Hall Object-Oriented Series. Prentice Hall International, Hemel Hempstead, UK, 1992. 8, 10
18. B. Meyer. *Object-Oriented Software Construction*. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, second edition, 1997. 3, 5, 8, 10
19. W. B. Mugridge, J. Hamer, and J. G. Hosking. Multi-methods in a statically-typed programming language. In P. America, editor, *Proc. ECOOP'91*, LNCS 512, pages 307–324. Springer-Verlag, 1991. 8
20. OMG. Unified Modeling Language specifications, v1.4. Technical report, Object Management Group, 2001. 10
21. D. Rayside and G. Campbell. An aristotelian understanding of object-oriented programming. In *Proc. OOPSLA'00*, SIGPLAN Notices, 35(10), pages 337–353. ACM Press, 2000. 3, 4
22. D. Rayside and K. Kontogiannis. On the syllogistic structure of object-oriented programming. In *Proc. of ICSE'01*, pages 113–122, 2001. 3, 4
23. J.-C. Royer. An Operational Approach to the Semantics of Classes: Application to Type Checking. *Programming and Computer Software*, 28(3), 2002. (to appear). 8
24. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1991. 10

25. C. Szypersky, S. Omohundro, and S. Murer. Engineering a programming language: The type and class system of Sather. In *Proc. of First Int. Conference on Programming Languages and System Architectures*, LNCS 782. Springer Verlag, 1994. 3, 9
26. F. Weber. Getting class correctness and system correctness equivalent — how to get covariant right. In R. Ege, M. Singh, and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems (TOOLS 8)*, pages 192–213, 1992. 9
27. W. Woods and J. Schmolze. The KL-ONE family. *Computers & Mathematics with Applications*, 23(2–5):133–177, 1992. 5

Maintaining Class Membership Information

Anne Berry and Alain Sigayret

LIMOS
Ensemble Scientifique des Cézeaux, Université Blaise Pascal
63 170 Aubière, France
berry@isima.fr
sigayret@isima.fr

Abstract. Galois lattices (or concept lattices), which are lattices built on a binary relation, are now used in many fields, such as Data Mining and hierarchy organization, but may be of exponential size. In this paper, we propose a decomposition of a Galois sub-hierarchy which is of small size but contains useful inheritance information. We show how to efficiently maintain this information when an element is added to or removed from the relation, using a dynamic domination table which describes the underlying graph with which we encode the lattice.

1 Introduction

Galois lattices (also called concept lattices), are an emerging tool in research areas such as Data Mining, Database Managing and Object Hierarchy Organization (see [5], [10], [11], [13], [14], [15]).

A lattice has the advantage over a tree that it allows a much more complex structure, as every pair of elements not only has a greatest lower bound, but also has a lowest upper bound. In particular, this structure has been shown to be well adapted to representing multiple inheritance (a car can be both a wheeled vehicle and water-faring).

Concept lattices, moreover, are built from a binary relation, classically between a set $\mathcal{P}$ of properties and a set $\mathcal{O}$ of objects, which completely represents the information which is to be analysed. The elements of the lattice describe all possible maximal associations of the properties and objects. They are both a powerful investigation tool for Data Mining applications, and a complete underlying structuration for relationships, which makes them a good basis for extracting an organization into hierarchies.

The main drawback of such a lattice is that it may be exponential in size compared to the initial binary relation it is constructed from. Though it is known that, when one can give an upper bound on the number of properties for an object, the lattice is of polynomial size (see [4], [6], [7]), there is no known *general* characterization for relations which will define only a polynomial number of concepts. As a result, users of concept lattices are left with few options:

– use only a part of the lattice, by defining a sub-lattice or applying zooming techniques;

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 13–23, 2002.

- use and maintain a relation which belongs to a class which is known to define only a polynomial number of concepts;
- use a polynomial representation of the lattice to extract the most pertinent information.

In this paper, we will investigate the third possibility, by using both an underlying polynomial-sized graph which we use to encode the lattice (see [2]) and by restricting the information to a variant of a Galois sub-hierarchy.

Moreover, we address the issue of maintaining such information when an element is added to or deleted from the relation, without re-computing the entire sub-hierarchy.

2 Concept Lattices and Galois Sub-hierarchies

Given a finite set $\mathcal{P}$ of "properties" (which may be attributes, methods, features, etc., and which we will denote by lowercase letters) and a finite set $\mathcal{O}$ of "objects" (which may be tuples, individuals, classes, etc., and which we will denote by numbers), we consider a binary relation R as a proper subset of the Cartesian product $\mathcal{P} \times \mathcal{O}$; we will refer to the triple $(\mathcal{P}, \mathcal{O}, R)$ as a **context**.

Given a context $C = (\mathcal{P}, \mathcal{O}, R)$, a **concept** or **closed set** of C, also called a **maximal rectangle** of R, is a sub-product $A \times B \subseteq R$ such that $\forall x \in \mathcal{O} - B, \exists y \in A \,|\, (y, x) \not\in R$, and $\forall x \in \mathcal{P} - A, \exists y \in B \,|\, (x, y) \not\in R$. A is called the **intent** of the concept, B is called the **extent**.

The set of concepts thus defined form a lattice when ordered by inclusion on the intents, or, dually, by inclusion on the extents, called a **concept lattice** or **Galois lattice**.

Example 1. $\mathcal{P} = \{a, b, c, d, e, f\}$, $\mathcal{O} = \{1, 2, 3, 4, 5, 6\}$. Binary relation R:

	a	b	c	d	e	f	g	h
1		×	×	×	×			
2	×	×	×				×	×
3	×	×				×	×	×
4				×	×			
5			×	×				
6	×							×

The associated concept lattice $\mathcal{L}(R)$ is shown in Figure 1.

Because of the inheritance rules associated with this lattice, the labels can be simplified into mentioning only properties or objects which occur for the first time, in a top-bottom fashion for properties, in a bottom-top fashion for objects.

Example 2. The simplified concept lattice obtained from $\mathcal{L}(R)$ associated with relation R of Example 1 is shown in Figure 1.

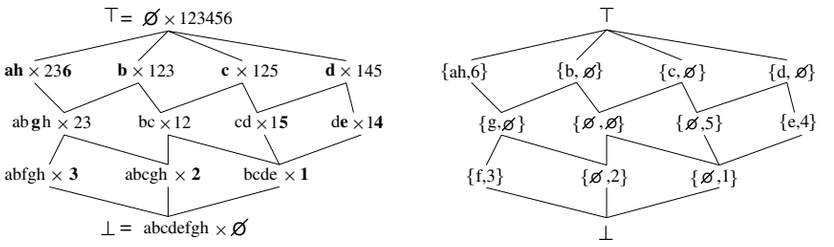


Fig. 1. Concept lattice $\mathcal{L}(R)$ and corresponding simplified lattice of relation R of Example 1

When the number of elements of the lattice is exponential, several authors ([5], [3], [10]) have found it useful to further simplify this lattice into a **Galois sub-hierarchy**, by defining a partially ordered set obtained by removing from this simplified lattice trivial nodes such as empty set pairs $\{\emptyset, \emptyset\}$, and usually the top and bottom elements.

Example 3. Figure 2 shows the Galois sub-hierarchy obtained from the simplified lattice of Example 2. Note that this partial ordering does not define a lattice, as elements $\{\emptyset, 2\}$ and $\{\emptyset, 1\}$ fail to have a unique nearest common descendent.

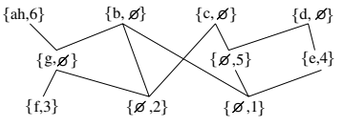


Fig. 2. Galois sub-hierarchy obtained from the simplified lattice of Example 2

3 Decomposing a Galois Sub-hierarchy Using Graph Domination

Our approach to encoding a Galois lattice (see [2]) is, surprisingly enough, to use an underlying graph G_R, constructed on the complement of the relation, defined, for a given context $(\mathcal{P}, \mathcal{O}, R)$ as $G_R = (V, E)$, with $V = \mathcal{P} \cup \mathcal{O}$, and with edges defined as:

1. internal edges which make $\mathcal{P}$ and $\mathcal{O}$ into cliques (if $x, y \in \mathcal{P}$ then $xy \in E$ and if $x, y \in \mathcal{O}$ then $xy \in E$).
2. external edges: if $x \in \mathcal{P}$ and $y \in \mathcal{O}$ then $xy \in E$ iff $(x, y) \notin R$.

Since $\mathcal{P}$ and $\mathcal{O}$ are trivially cliques, we will not represent their internal edges, nor will these have any influence on the complexity evaluations we discuss, as they need not be traversed. We will denote by $N^+(x)$ the external neighborhood of vertex x: if $x \in \mathcal{P}, N^+(x) = \{y \in \mathcal{O} | (x,y) \not\in R\}$, and if $x \in \mathcal{O}, N^+(x) = \{y \in \mathcal{P} | (y,x) \not\in R\}$. We use $n = |\mathcal{P}| + |\mathcal{O}|$, and $m = |\mathcal{P}| \times |\mathcal{O}| - |R|$.

Note that, though not much is known on the size of the concept lattice defined by a given relation, in general the lattice tends to be exponential in size when it is dense (i.e. when it has many crosses); in this case, for our graph, m will be of the order of n instead of n^2.

The reason we define this graph is that we have the remarkable property that a vertex set S of G_R is a minimal separator of G_R, separating connected component A from connected component B if, and only if $A \times B$ is a concept defined by relation R. Now this may seem a little far-fetched, but a steady output of work done in the past decade has yielded many results on minimal separation, and in [2] we show this to be an efficient tool for concept lattice investigation and concept generation. It is interesting to note, however, that quite regularly work appears on the relationship between graphs and lattices (see [1], [8], [9], [12]).

One of the related graph notions which turns out to be of primary importance for the study of concept lattices is that of **domination**: a vertex x is said to dominate another vertex y if $N^+(y) \subset N^+(x)$.

The domination relation defines a partial pre-ordering on V, which we decompose into the **property domination relation** and the **object domination relation**.

Example 4. The relation of Example 1 yields the graph given in Figure 3. $N^+(a) = \{1,4,5\}$, $N^+(b) = \{4,5,6\}$, $N^+(c) = \{3,4,6\}$, $N^+(d) = \{2,3,6\}$, $N^+(e) = \{2,3,5,6\}$, $N^+(f) = \{1,2,4,5,6\}$, $N^+(g) = \{1,4,5,6\}$, $N^+(h) = N^+(a)$, $N^+(1) = \{a,f,g,h\}$, $N^+(2) = \{d,e,f\}$, $N^+(3) = \{c,d,e\}$, $N^+(4) = \{a,b,c,f,g,h\}$, $N^+(5) = \{a,b,e,f,g,h\}$, $N^+(6) = \{b,c,d,e,f,g\}$.

a and h share the same neighborhood and behave as a single vertex ah. f dominates g, g dominates a and b; by transitivity, this implies that f also dominates a and b. c is neither dominated nor dominating.

Figure 3 gives the property domination relation and the object domination relation.

It is clear from Example 4 that this domination relation is strongly related to the Galois sub-hierarchy shown in Figure 2.

In order to precisely describe this relationship, we introduce a decomposition of the Galois sub-hierarchy into the sub-hierarchy of intents and the sub-hierarchy of extents, by using only the left or right parts of the labels of the sub-hierarchy. If, consistently with the original definition, we then remove the empty set nodes, we find the domination relation.

Example 5. Figure 4 gives the sub-hierarchies of intents and of extents derived from Figure 2.

Based on these considerations, we give the following property:

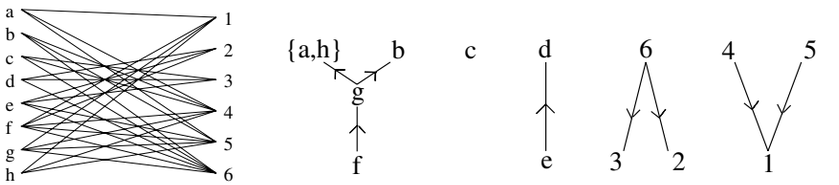


Fig. 3. Graph G_R coding the relation from Example 1 and the corresponding domination relation

Fig. 4. Sub-hierarchies of intents and of extents derived from Figure 2

Property 1. The property domination relation of G_R is equivalent to the Galois sub-hierarchy taken on the intents, from which all empty set nodes have been removed, and the object domination relation of G_R is equivalent to the Galois sub-hierarchy taken on the extents, from which all empty set nodes have been removed.

4 Computing and Updating the Domination Information

Computing the domination relation of a graph costs roughly $O(nm)$ time. In this section, we will introduce a data structure which will enable us to update a relation by adding or deleting elements, with a cost of only $O(n)$ per update.

We will restrict our description of our process to computing the property domination relation; of course, the same process can be applied dually to computing the object domination relation.

4.1 Computing and Querying the Domination Table

In order to maintain property domination information, we construct a domination table, which, for each pair (x, y) of properties, lists the objects, the presence of which prevents x from dominating y, or from having a neighborhood which is the same as that of y. This just means that if for object i, $(x, i) \in R$ and $(y, i) \not\in R$, i will appear in the list for (x, y).

Example 6. Property domination table corresponding to relation R from Example 1:

	a	b	c	d	e	f	g	h
a	$\emptyset$	$\{1\}$	$\{1,5\}$	$\{1,4,5\}$	$\{1,4\}$	$\emptyset$	$\emptyset$	$\emptyset$
b	$\{6\}$	$\emptyset$	$\{5\}$	$\{4,5\}$	$\{4\}$	$\emptyset$	$\emptyset$	$\{6\}$
c	$\{3,6\}$	$\{3\}$	$\emptyset$	$\{4\}$	$\{4\}$	$\{3\}$	$\{3\}$	$\{3,6\}$
d	$\{2,3,6\}$	$\{2,3\}$	$\{2\}$	$\emptyset$	$\emptyset$	$\{3\}$	$\{2,3\}$	$\{2,3,6\}$
e	$\{2,3,6\}$	$\{2,3\}$	$\{2,5\}$	$\{5\}$	$\emptyset$	$\{3\}$	$\{2,3\}$	$\{2,3,6\}$
f	$\{2,6\}$	$\{1,2\}$	$\{1,2,5\}$	$\{1,4,5\}$	$\{1,4\}$	$\emptyset$	$\{2\}$	$\{2,6\}$
g	$\{6\}$	$\{1\}$	$\{1,5\}$	$\{1,4,5\}$	$\{1,4\}$	$\emptyset$	$\emptyset$	$\{6\}$
h	$\emptyset$	$\{1\}$	$\{1,5\}$	$\{1,4,5\}$	$\{1,4\}$	$\emptyset$	$\emptyset$	$\emptyset$

This table is to be read as (x, y) pairs, each containing the information on x dominating y, where x labels a column and y a row.

The algorithmic process for constructing the property domination table is simple:

For each $x \in \mathcal{P}$
  For each $y \in \mathcal{P}$
    For each $z \in \mathcal{O}$
      If $(x, z) \in R$ and $(y, z) \notin R$ then add z to list (x, y);

The process for building the object domination table is symmetric, and obtained from the previous one by exchanging $\mathcal{P}$ and $\mathcal{O}$.

The global size of the tables is of $O(nm)$; as each entry can be computed in constant time using relation R, the total time cost for computing the property domination table and the object domination table will be in $O(nm)$.

From the table, we can deduce that:

- a will dominate b if object 6 is deleted.
- e dominates d since $(e, d) = \emptyset$. The property domination relation shown in Figure 3 can easily be globally computed by examining this domination for each property.
- $N^+(a) = N^+(h)$, since a dominates h and h dominates a; note how columns a and h are identical.
- $N^+(g) = N^+(a) \cup N^+(b)$, as column g is the intersection of columns a and b in the domination table. This means that relation R fails to be a reduced relation: g appears in the intent of a concept of $\mathcal{L}(R)$ iff both a and b appear too.

Computing the intents and extents of the sub-hierarchy elements from the domination table.
For a given property, for example f, find from the table which properties f dominates, that is which have $\emptyset$ in the f column; this query yields a, b, f, g, h; $abfgh$ will be the intent; then compute $\mathcal{O} - N^+(f) = \{3\}$, which yields the extent of the element; we thus obtain element $abfgh \times 3$ as element of $\mathcal{L}(R)$, in a complexity proportional to the size of the result.

Computing the simplified Galois sub-hierarchy from the domination table.
Compute as above the intent which corresponds to each property x; tentatively store $\{x, \emptyset\}$ as an element of the sub-hierarchy; for each object, likewise compute the corresponding intent; if this intent has already been computed by some property x, then replace $\{x, \emptyset\}$ with $\{x, y\}$ as an element of the sub-hierarchy; else add element $\{\emptyset, y\}$ to the list of elements of the sub-hierarchy. For example, property 3 will yield intent $\mathcal{P} - N^+(3) = \{a, b, f, g, h\}$, which has already been computed by f: replace $\{f, \emptyset\}$ with $\{f, 3\}$ as element of the simplified sub-hierarchy. This requires roughly $O(n^2)$ time, but this complexity can be streamlined.

4.2 Updating the Domination Table

1. When **adding an element** (x, z) to relation R, with $x \in \mathcal{P}$ and $z \in \mathcal{O}$, which means adding a cross in R at location (x, z), or, equivalently, removing edge xz from the corresponding coding graph G_R:
   - for each "non-cross" y of line z in R (i.e. $(y, z) \notin R$), **add** z to list (x, y);
   - for each "cross" y of line z in R (i.e. $(y, z) \in R$), **delete** z from list (y, x);
2. When **deleting an element** (x, z) from relation R, $x \in \mathcal{P}, z \in \mathcal{O}$:
   - for each "non-cross" y of line z in R (i.e. $(y, z) \notin R$), **delete** z from list (x, y);
   - for each "cross" y of line z in R (i.e. $(y, z) \in R$), **add** z to list (y, x);

Updating the table will cost time $O(|\mathcal{P}|)$.

Example 7. Let us add element $(b, 5)$ to relation R of Example 1.

New relation R' obtained:

	a	b	c	d	e	f	g	h
1		×	×	×	×			
2	×	×	×				×	×
3	×	×				×	×	×
4				×	×			
5		×	×	×				
6	×							×

Line 5 contains non-crosses a, e, f, g, h and crosses c, d. 5 must be added to the lists of (b, a), (b, e), (b, f), (b, g) and (b, h), and deleted from the lists of (c, b) and (d, b).

New domination table obtained:

	a	b	c	d	e	f	g	h
a	$\emptyset$	$\{1, \mathbf{5}\}$	$\{1,5\}$	$\{1,4,5\}$	$\{1,4\}$	$\emptyset$	$\emptyset$	$\emptyset$
b	$\{6\}$	$\emptyset$	$\{\not{5}\}$	$\{4, \not{5}\}$	$\{4\}$	$\emptyset$	$\emptyset$	$\{6\}$
c	$\{3,6\}$	$\{3\}$	$\emptyset$	$\{4\}$	$\{4\}$	$\{3\}$	$\{3\}$	$\{3,6\}$
d	$\{2,3,6\}$	$\{2,3\}$	$\{2\}$	$\emptyset$	$\emptyset$	$\{3\}$	$\{2,3\}$	$\{2,3,6\}$
e	$\{2,3,6\}$	$\{2,3,\mathbf{5}\}$	$\{2,5\}$	$\{5\}$	$\emptyset$	$\{3\}$	$\{2,3\}$	$\{2,3,6\}$
f	$\{2,6\}$	$\{1,2,\mathbf{5}\}$	$\{1,2,5\}$	$\{1,4,5\}$	$\{1,4\}$	$\emptyset$	$\{2\}$	$\{2,6\}$
g	$\{6\}$	$\{1,\mathbf{5}\}$	$\{1,5\}$	$\{1,4,5\}$	$\{1,4\}$	$\emptyset$	$\emptyset$	$\{6\}$
h	$\emptyset$	$\{1,\mathbf{5}\}$	$\{1,5\}$	$\{1,4,5\}$	$\{1,4\}$	$\emptyset$	$\emptyset$	$\emptyset$

As a result of the modification of R, c now dominates b. Figure 5 shows the new concept lattice $\mathcal{L}(R')$ and the associated Galois sub-hierarchy; Figure 6 gives the new property domination relation.

Let us now delete element $(b, 1)$ from the previous relation R'.

New relation R'' obtained:

	a	b	c	d	e	f	g	h
1			×	×	×			
2	×	×	×				×	×
3	×	×				×	×	×
4				×	×			
5		×	×	×				
6	×							×

Line 1 contains non-crosses a, f, g, h and crosses c, d, e. 1 must be deleted from the lists of $(b, a), (b, f), (b, g)$ and (b, h), and added to the lists of $(c, b), (d, b)$ and (e, b).

New domination table obtained:

	a	b	c	d	e	f	g	h
a	$\emptyset$	$\{\not{1}, 5\}$	$\{1,5\}$	$\{1,4,5\}$	$\{1,4\}$	$\emptyset$	$\emptyset$	$\emptyset$
b	$\{6\}$	$\emptyset$	$\{\mathbf{1}\}$	$\{\mathbf{1}, 4\}$	$\{\mathbf{1}, 4\}$	$\emptyset$	$\emptyset$	$\{6\}$
c	$\{3,6\}$	$\{3\}$	$\emptyset$	$\{4\}$	$\{4\}$	$\{3\}$	$\{3\}$	$\{3,6\}$
d	$\{2,3,6\}$	$\{2,3\}$	$\{2\}$	$\emptyset$	$\emptyset$	$\{3\}$	$\{2,3\}$	$\{2,3,6\}$
e	$\{2,3,6\}$	$\{2,3,5\}$	$\{2,5\}$	$\{5\}$	$\emptyset$	$\{3\}$	$\{2,3\}$	$\{2,3,6\}$
f	$\{2,6\}$	$\{\not{1}, 2,5\}$	$\{1,2,5\}$	$\{1,4,5\}$	$\{1,4\}$	$\emptyset$	$\{2\}$	$\{2,6\}$
g	$\{6\}$	$\{\not{1}, 5\}$	$\{1,5\}$	$\{1,4,5\}$	$\{1,4\}$	$\emptyset$	$\emptyset$	$\{6\}$
h	$\emptyset$	$\{\not{1}, 5\}$	$\{1,5\}$	$\{1,4,5\}$	$\{1,4\}$	$\emptyset$	$\emptyset$	$\emptyset$

After this second modification of R, c does not dominate b any longer. Figure 7 shows the new concept lattice and the associated Galois sub-relation.

The property domination relation reverts to its original form, given by Figure 3, even though the lattice is larger and structurally significantly different.

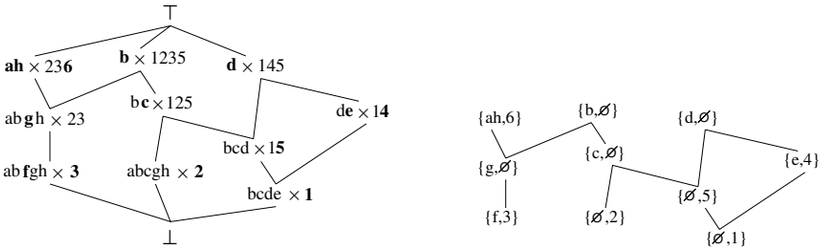


Fig. 5. New concept lattice $\mathcal{L}(R')$ and the associated Galois sub-hierarchy obtained when adding $(b, 5)$ to relation R of Example 1

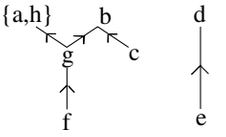


Fig. 6. Domination relation obtained when adding $(b, 5)$ to relation R of Example 1

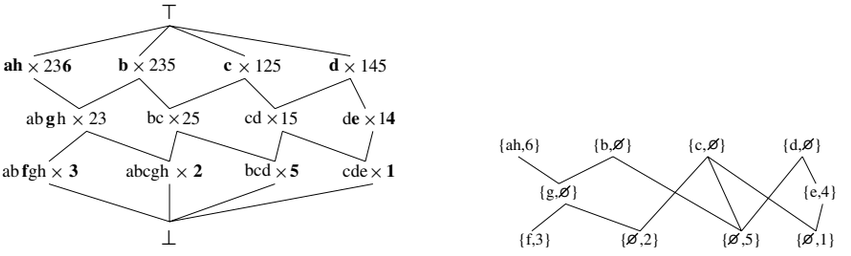


Fig. 7. New concept lattice $\mathcal{L}(R'')$ and the associated Galois sub-hierarchy obtained when deleting $(b, 1)$ from the new relation R' of Example 7

5 Conclusion

We have shown a new approach to modifying significant information we need to extract from a concept lattice structure, with an efficient updating technique, based on a new graph-based data structure which enables us to avoid re-computing the entire structure.

This updating technique could be extended to efficiently constructing the initial domination table for a given relation which is very dense, by first considering the relation with only a diagonal of zeroes, which describes a graph with no domination, and then removing elements from this until the desired relation is obtained.

Conversely, the principle of domination table could also be used to model a relation into respecting a given sub-hierarchy, with particular desirable relationships between classes.

Acknowledgment

We thank the referees for their very interesting questions and suggestions.

References

1. Berry, A., Bordat, J.–P.: Orthotreillis et séparabilité dans un graphe non-orienté. Mathématiques, Informatique et Sciences Humaines, **146** (1999) 5–17. 16
2. Berry, A., Sigayret, A.: Representing a concept lattice by a graph. Workshop on Discrete Mathematics for Data Mining, Proc. 2nd SIAM Workshop on Data Mining, Arlington (VA), April 11–13, (2002). 14, 15, 16
3. Chen, J.–B., Lee, S. C.: Generation and Reorganization of Subtype Hierarchies. Journal of Object Oriented Programming, **8(8)** (1996). 15
4. Godin, R.: Complexité de Structures de Treillis. Annales des Sciences Mathématiques du Québec, **13(1)** (1989) 19–38. 13
5. Godin, R., Mili, H.: Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices. Proceedings of ACM OOPSLA'93, Special issue of Sigplan Notice, **28(10)** (1993) 394–410. 13, 15
6. Godin, R., Missaoui, R., April, A.: Experimental Comparison of Navigation in a Galois Lattice with Conventional Information Retrieval Methods. International Journal of Man-Machine Studies, **38** (1993) 747–767. 13
7. Godin, R., Saunders, E., Gecsei, J.: Lattice Model of Browsable Data Spaces. Information Sciences, **40** (1986) 89–116. 13
8. Hager, M.: On Halin-Lattices in Graphs. Discrete Mathematics, **47** (1983) 235–246. 16
9. Halin, R.: Lattices of cuts in graphs. Abh. Math. Sem. Univ. Hamburg, **61** (1991) 217–230. 16
10. Huchard, M., Dicky, H., Leblanc, H.: Galois lattice as a framework to specify building class hierarchies algorithms. Theoretical Informatics and Applications, **34** (2000) 521–548. 13, 15

11. Pfaltz, J. L., Taylor, C. M.: Scientific Knowledge Discovery through Iterative Transformation of Concept Lattices. Workshop on Discrete Mathematics for Data Mining, Proc. 2nd SIAM Workshop on Data Mining, Arlington (VA), April 11–13, (2002). 13
12. Polat, N.: Treillis de séparation des graphes. Can. J. Math., vol. **XXVIII** No **4** (1976) 725–752. 16
13. Valtchef, P., Missaoui, R., Godin, R.: A Framework for Incremental Generation of Frequent Closed Item Sets. Workshop on Discrete Mathematics for Data Mining, Proc. 2nd SIAM Workshop on Data Mining, Arlington (VA), April 11–13, (2002). 13
14. Yahia, A., Lakhal, L., Bordat, J.–P.: Designing Class Hierarchies of Object Database Schemes. Proceedings 13e journées Bases de Données avancées (BDA'97), (1997). 13
15. Zaki, M. J., Parthasarathy, S., Ogihara M., Li, W.: New Algorithms for Fast Discovery of Association Rules. Proceedings of 3rd Int. Conf. on Database Systems for Advanced Applications, April, (1997). 13

Hierarchies in Object Oriented Conceptual Modeling

Esperanza Marcos and Jose María Cavero

Kybele Research Group, Rey Juan Carlos University
28933 Móstoles (Madrid), Spain
{cuca,j.m.cavero}@escet.urjc.es

Abstract. One of the most powerful tools of abstraction used in object-oriented conceptual modelling is the specialisation/generalisation hierarchy, which allows representing taxonomic relationships among classes. A specialisation/generalisation hierarchy (from now on, taxonomy or taxonomic hierarchy) has always two main associated characteristics: the classification and the inheritance concepts. There are, however, different kinds of taxonomic hierarchies (classification, inheritance, role, etc.) which are often confused or misused in modelling. Sometimes, this occurs because one kind of taxonomic hierarchy is used to represent two or more different concepts. In this paper we present a model of useful taxonomic hierarchies for conceptual modelling from an object-oriented perspective. The taxonomic hierarchies that we propose to represent knowledge are based on Aristotle's definition of essence and accident.

1 Introduction

One of the most important objectives in conceptual modelling consists in narrowing the gap between the real world and its representation. To make this possible, conceptual models have improved their expressiveness through new primitives, which are closer to the real world. The specialisation/generalisation hierarchy (from now on, *taxonomy* or *taxonomic hierarchy*) is generally considered to be one of the most important structures of reasoning and knowledge representation. Every object-oriented conceptual model supports taxonomic hierarchies as a natural way of representing data. However, the name and meaning can vary in different models: specialisation inheritance [4, 12], classification [7], IS-A generalisation [9] etc. For the rest of the paper we define **taxonomy** or **taxonomic hierarchy** as:

> "Hierarchy of classes that allows us to classify the objects of a class by specialising or generalising them. A taxonomy should satisfy two properties: a) classification: the extension of every subclass is a subset of the extension of the superclass; b) inheritance: the subclass inherits all properties (attributes, methods, restrictions, etc.) of the superclass"

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 24-33, 2002.

It is important to underline that inheritance is not a taxonomy but a characteristic of the taxonomy. However, at the implementation level (that is, in programming, not in conceptual modelling), we could define hierarchies that only satisfy the property of inheritance. The only purpose of this kind of hierarchies is to share and reuse code. We shall use ***inheritance hierarchy*** in reference to *"hierarchies that do not satisfy the principle of classification"*. ***Inheritance*** is a property of every hierarchy (taxonomic or inheritance hierarchy) while ***classification*** is just a property of the taxonomic hierarchy.

One of the main problems of using taxonomic hierarchies in conceptual modelling is that the concept of taxonomy lumps together a variety of representation mechanisms (suppose, for example, the problems that could arise if we had just one constructor to represent objects and relationships, or objects and attributes). There is also some confusion among different concepts of classification and between classification and inheritance [1], which may be one of the reasons why some authors [10, 14] consider it very difficult to find examples of *inheritance hierarchies* outside the realm of biology. The problem arises because we always consider that any identified taxonomy is similar to its biological referents. Aristotle's biological taxonomy is an important classification because it has been used as a model for other classifications. However, it cannot be considered as the universal taxonomic model, because it is not directly applicable to every classification in the world. In [1], the authors posit the need to integrate the two views of inheritance, i.e., conceptual modelling and code reuse.

As stated by [13], if conceptual modelling constructors[1] are to be used effectively, their meanings will have to be defined rigorously. Since conceptual models are intended to capture knowledge about a real world domain, we consider that the meaning of the model constructors should be sought in models of reality. In this paper, we analyse the use of multiple taxonomies in conceptual modelling. The identified taxonomies are based on Aristotle's definition of essence *vs*. accident and change *vs*. motion.

The rest of the paper is organised as follows: section 2 is an overview of the different kinds of taxonomies that we have identified to use in conceptual modelling; in section 3 we analyse the possibilities of defining multiple taxonomies in conceptual modelling; section 4 poses the main conclusions and future work.

2 A Classification of Hierarchies

One of the main problems speaking of taxonomies is that there is no general agreement as to what is and what is not a taxonomic hierarchy. Different authors identify different kinds of taxonomic hierarchies, with different names, and with different characteristics and requirements [2, 3, 5, 8, 11, 14]. This is because the classifications of taxonomies proposed in the literature are based on taxonomies defined in different models (conceptual or implementation models). Our approach is radically opposite:

[1] Conceptual modelling constructors and object oriented programming constructors are not the same. For example, the aggregation and association are conceptual modelling constructors that have usually the same representation at implementation level.

we seek to classify taxonomic hierarchies according to the ontology of the world that we want to model, and we will characterise these taxonomies to determine how conceptual (and implementation) models should represent them [6].

In the world, taxonomies, as well as classes, serve to classify objects according to different criteria. Classes classify objects that have similar characteristics and similar behaviours, while taxonomies offer a more specific classification, for we can specialise, or generalise, the objects of a specific class. A taxonomy should verify that the extension of the subclass is a subset of the extension of the superclass. Our approach is based on Aristotle's ontology.

2.1 Aristotle's Ontology

As we have said, our approach is based on Aristotle's ontology, particularly on the distinction between substance and accident, and also on the change and motion concepts. According to Aristotle, things are substances with accidents (Metaphysics 025). He also distinguishes between *substantial change* that depends on the essence, and *accidental change* that depends on some accidental characteristic (Physics V). Whereas the accidental change implies any motion on the individual, there is no motion in the substantial change. He stated that there are only three kinds of change: from subject to non-subject (perishing), from subject to subject (movement) and from non-subject to subject (becoming) (Physics V 225a). Every perishing or becoming change implies a substance change, whereas a motion implies an accidental change. Table 1 summarises Aristotle's ontology in relation with the object-oriented ontology.

An essential characteristic is a characteristic that determines the substance of the object. If the value of an essential characteristic change, the object changes. However, the value of an accidental characteristic can change and the object continues being the same. We have identified three different kinds of taxonomies depending if the criterion chosen to build the taxonomy is based on an *essential* or *accidental* characteristic: IS-A, STATE and ROLE taxonomy. The IS-A taxonomy is built according to the value of some essential characteristic of the objects (that is to say, if the value of the characteristic changes, the object changes). The rest of the taxonomies are based on accidental characteristics, whose value can change while the object continues being the same.

Table 1. Aristotle's ontology

Aristotle's ontology	Object Oriented Conceptual Model-
Subject	Object
Substance	Attribute that is the essence of the
Accident	Attribute whose value can change
Accidental change	Change the value of an attribute
Motion	Migration
Substantial change	Change of the essence
Perishing	Delete object
Becoming	Create object

2.2 The Hierarchies

As we have already said, three possible taxonomic hierarchies can be considered as constructors for conceptual modelling. To define them and for the rest of the paper we consider the following definitions:

- An *object* is an instance of a class. It is a *subject* in Aristotle's terminology.
- The *extension* of a class C, denoted as ext(C), is the set of objects belonging to the class at a given state.
- The *intension* of a class is the set of all the properties shared by possible objects of the class.
- C1 is a *subclass* of C0, if ext(C1) is a subset of ext(C0), for every possible state.
- An object *migrates* when it moves from one class to another, in the same level of the hierarchy, or in another level but in a different branch of the tree. That is to say, the target class can be neither an ancestor nor a descendant of the original class. If the object moves to any class in another level of the hierarchy, then the object is specialised or generalised.

IS-A Taxonomy

The IS-A taxonomy classifies objects in different classes according to some essential characteristic of these objects. In a less strict classification, these objects could belong to a single class. Taking the biological taxonomy example, see figure 1(a), human is a class that specialises the mammal class according to an essential characteristic: the species. Figure 1(b) shows an extension of the example in figure 1(a), where John, Peter, Lassie and Milky represent objects. For this kind of taxonomy, we define three rules:

Rule 1. The objects of the subclasses also belong to the superclass. That is to say, the subclass is always a subset of the superclass, as can be seen in figure 1(b).

Rule 2. Subclasses are mutually disjoint. This is because the classification is made with regard to an essential characteristic, and it is impossible for an essential characteristic of an object to have two different values simultaneously. For example, a mammal cannot be simultaneously a human and a bovine (see figure 1(b)), because it cannot belong to more than one specie.

Rule 3. Object migration is not allowed, because if the essential feature is changed, the object will not be the same. In our example, a human cannot become a cow.

As we have already said, unlike other classifications of taxonomic hierarchies, our proposal is based on the semantics of the world, but this approach has a shortcoming: the very subjectivity of the world. Sometimes it is not clear what the essential characteristic of a thing actually is. In our example (we have looked for an unquestionable one) it seems quite clear that a person and a cow are essentially different from each other. It is impossible for "John" to become a cow. People who believe in reincarnation, however, consider it possible for a person to come back as a cow without changing their essence. In spite of this philosophical issue, we believe that an essential criterion can always be found, albeit an arbitrary one, as is sometimes the case in conceptual modelling for distinguishing between attributes and classes.

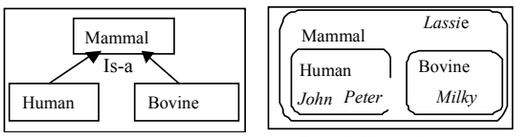


Fig. 1(a)- Example of an IS-A Fig. 1(b)- Disjoint subclasses

Fig. 1. An example of the IS-A taxonomy

ROLE Taxonomy

The ROLE taxonomy puts objects in different classes, according to the role each one plays. In a less strict classification, these objects could belong to the same class. Figure 2 shows an example of the ROLE taxonomy. For this kind of taxonomy, we define three rules:

Rule 1. The subclass is always a subset of the superclass
Rule 2. Unlike the IS-A hierarchy, subclasses are not mutually disjoint (see figure 2 (b)). The same object can be classified in two different classes, according to two different roles played simultaneously. In our example, a man can simultaneously play the roles of father and husband.
Rule 3. Object migration is allowed. Classification is based on the role rather than on any essential characteristics, and the role can change at any moment.

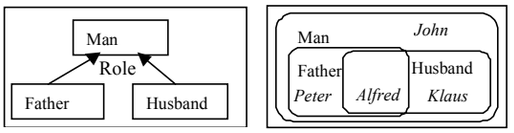


Fig. 2(a)- Example of a ROLE taxonomy Fig. 2(b)- Overlapping subclasses

Fig. 2. An example of the ROLE taxonomy

STATE Taxonomy

The STATE taxonomy classifies objects according to their state at a given moment. In a less strict classification, such objects could belong to a single class. Figure 3 shows an example of the STATE taxonomy. We define three rules:

Rule 1. The subclass is always a subset of the superclass.
Rule 2. Subclasses are mutually disjoint, for classification is made with regard to mutually exclusive states. For example, see figure 3 (b), a person cannot be married and single at the same time.
Rule 3. Object migration is allowed. Classification is based on the state and the state can change at any moment.

As in the previous sections, exceptions to our rule can be found. For example, a person can be legally single in one country and married in another. We shall therefore also have to define an arbitrary group of states. The chosen state, as is the case with the role and the essential characteristic, will depend on the universe of discourse.

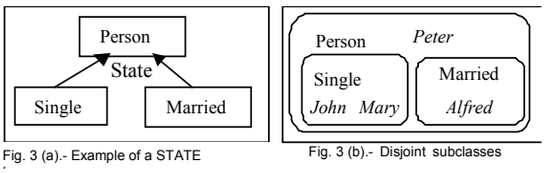


Fig. 3 (a).- Example of a STATE

Fig. 3 (b).- Disjoint subclasses

Fig. 3. An example of the STATE taxonomy

The inheritance hierarchy is not a taxonomic hierarchy but a characteristic of the taxonomic hierarchies. However, it is very common to define hierarchies just to share properties. The next section defines inheritance hierarchies.

Inheritance Hierarchy

The inheritance hierarchy does not classify objects. It is just a relationship between classes that allows objects to share properties. The main difference with the proposed taxonomies is that, in a less strict classification, such objects would not normally belong to a single class, i.e. inheritance hierarchy is not a taxonomic hierarchy. So, the classification principle is not applicable. Figure 4(a) shows an inheritance hierarchy example. For this kind of hierarchy, we formulate four rules:

Rule 1. The subclass is not necessarily a subset of the superclass. However, subclass characteristics do be a subset of the superclass.
Rule 2. Usually, subclasses are mutually disjoint. Because this hierarchy is not a classification, the objects of a subclass do not usually belong to another one. For example, see figure 4 (b), a company can never be a dog.
Rule 3. For the same reasons, object migration is not allowed.

The inheritance hierarchy is defined as an inclusion of properties rather than an inclusion of objects. Usually, subclasses are mutually disjoint and migration is not allowed.

This inheritance hierarchy does not represent a taxonomy. In the world, objects are not classified according to the name of their characteristics, because two different classes can share the same name for some characteristic without these characteristics being mutually equivalent. These characteristics have a different meaning, because they represent different things. For example, the name of a company does not represent the same thing as the name of a man. Thus, the only reason for using inheritance taxonomy is shared properties (especially behavioural properties, which allow for code reuse) and this is a design or implementation question rather than a conceptual one. There are therefore no semantic reasons for considering the inheritance hierarchy as a constructor for conceptual modelling.

Table 2 shows an overview of our classification, distinguishing between taxonomy hierarchies and their characteristics. We can observe that the differences between IS-A, ROLE and STATE taxonomies are based on the disjoint and migration characteristics. However the differences between inheritance hierarchy and taxonomic hierarchies is that the first one does not fulfil the subclass property that is mandatory in every taxonomic hierarchy. The similarity between inheritance hierarchy and taxonomic hierarchies is that all of them fulfil the inheritance property.

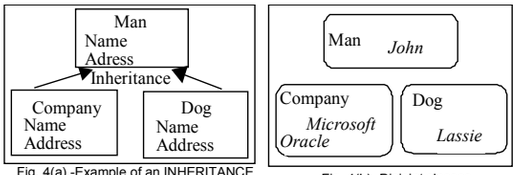


Fig. 4. An example of Inheritance Hierarchy

Table 2. Classification of Taxonomic Hierarchies

Characteristics		Inheritance	Subclasses	Migration	Disjoint
Non Taxonomic	**Taxonomic**				
	IS-A	Yes	Yes	No	Yes
	ROLE	Yes	Yes	Yes	No
	STATE	Yes	Yes	Yes	Yes
INHERIT.		Yes	No	No	Yes

2.3 The Object Oriented Hierarchies vs. Aristotle's Ontology

To sum up, we have identified three kinds of taxonomic hierarchies that should be used in conceptual modelling according to the ontology of the world that we want to represent. Our approach, as we have said, is based on Aristotle's philosophy, particularly on the distinction between substance and accident, and also on the change and motion concepts.

As an *IS-A taxonomy* is based on any essential characteristic of the objects it is only possible creating or deleting (becoming or perishing in Aristotle's terminology) objects (subjects), but it is impossible to move objects from a class to another (migration is not allowed). Moving objects from a class to another, due to the nature of the IS-A taxonomy, would imply a substantial change from object (subject) to object (subject) and, apart from being inconsistent with Aristotle's theory, the target object would not be the same object.

Both, *role and state taxonomies*, are based on Aristotle's accidental characteristics. As we have explained in section 2.1, there are only three kinds of changes and only the change from subject to subject, which is an accidental one, is a motion. Thus, an accidental change (that is, a role or a state change) can give rise to an object motion that is to an object migration. However, and due to the fact that perishing and becoming are substantial changes, an accidental change can never produce an instantiation or deletion.

We can also notice that the *inheritance hierarchy* has not any relationship with Aristotle's Ontology. The reason, again, is that inheritance is not a taxonomy and its task is not to classify but just to reuse code. Table 3, shows a comparison between the object oriented hierarchies and Aristotle's ontology.

The identified taxonomies are near to the **conceptual mechanisms** that we have to represent knowledge. These concepts are represented in English with just a verb: "to be". Although we have the three concepts in our minds, conceptual models, as English

language, do not provide us a way to distinguish among them. However, there are some languages that distinguish explicitly among some of these concepts. So, for example, the verb "to be" can be translated to the Spanish language in two different ways: "ser" (that refers to the IS-A concept) and "estar" (that means to be in a state). In fact, *IS-A* is translated to Spanish as *ES-UN* instead of *ESTA-EN-UN*. The Spanish verbs "ser" and "estar", allow us to make the difference between "to be A drunk" or "to be drunk". ROLE taxonomies can be identified because the verb "to be" can be replaced by another verb as "to work". For example "he is a teacher" can be replaced by "he works as a teacher".

Table 3. Object oriented hierarchies and Aristotle's ontology

Aristotle's Ontology		Kind of	King of	Movement allowed
Non Taxo-	Taxo-	attrib-	change	
	IS-A	Essen-	Substantial	Perishing, Becoming
	ROLE	Acci-	Accidental	Perishing, Becoming,
	STATE	Acci-	Accidental	Perishing, Becoming,
INHERIT.		-	-	-

3 Multiple Inheritance in Conceptual Modelling

When a class has two or more superclasses and the extension of the class is a subset of the extension of its superclasses (classification property), we have a multiple taxonomy. The focus of this discussion is on multiple taxonomies where the subclass inherits from two parents that belong to the same hierarchy, see figure 5(a), rather than on multiple parallel taxonomies, see figure 5(b).

Multiple taxonomy always involves multiple inheritance and multiple classification. So, multiple taxonomies will not be allowed when the supertypes belong to the same hierarchies and they are mutually disjoint. Thus, neither IS-A nor STATE taxonomies can be specialised by multiple taxonomies, unless they are specialised through inheritance hierarchies, which do not fulfil the classification property. ROLE taxonomies are not disjoint, and they can be specialised through: a multiple is-a taxonomy, a multiple role taxonomy, a multiple state taxonomy or a multiple inheritance hierarchy.

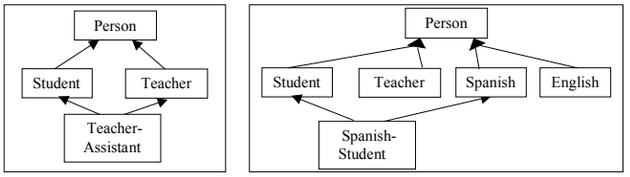


Fig. 5(a)- Example of multiple taxonomy

Fig. 5(b)- Example of multiple paralell taxonomies

Fig. 5. Multiple taxonomies and multiple parallel taxonomies

Any kind of taxonomic hierarchy can be specialised through multiple inheritance. But, as argued in section 2.4, there are no semantic reasons for considering inheritance hierarchy as a constructor for conceptual modelling. The only reason for using the

inheritance taxonomy is shared properties and this is a design or implementation decision rather than a conceptual one. Table 4 shows a summary of the possible combinations of multiple inheritance at conceptual level.

Table 4. Possible combinations of multiple inheritance

Subclass Superclasses	IS-A	ROLE	STATE	INHERITANCE
IS-A		Al-		Allowed
ROLE		Al-		Allowed
STATE		Al-		Allowed
INHERITANCE		Al-		Allowed

4 Conclusions

In this paper we have tried to clarify the meaning of the different taxonomic hierarchies that appear in the literature (often with different names and different meanings). The different taxonomic hierarchies are often used as if they were the same. So, it is possible to introduce an inheritance hierarchy in conceptual modelling. A proper definition of the different kinds of taxonomic hierarchies will also allow us to properly use taxonomic hierarchies in conceptual modelling. Figure 6 shows the classification of hierarchies that we have defined in this work and the relationship between them using our taxonomic model.

Future studies include: (1) studying these issues in multiple parallel taxonomies (for example, a person can be specialised according to the role he plays and according to his state simultaneously); (2) application of the theoretical conclusions to the conceptual modelling task.

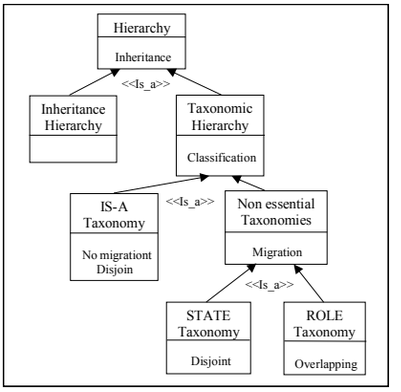


Fig. 6. Classification of hierarchies

Acknowledgements

This work is being carried out in the framework of the MIDAS project partially supported by the URJC (PGRAL -2001/05).

References

1. Al-Ahmad, W. and Steegmans, E. (1999), Modeling and Reuse Perspectives of Inheritance Can be Reconciled. In Proc. of the Technology of Object Oriented Languages and Systems (TOOLS'99) Ed. Chen, Lu and Meyer, IEEE, pag. 31-41.
2. Atkinson, M.P., Bancilhon, F., Dittrich, K.R., Maier, D. and Zdonik, S. (1989), The object-oriented database system manifesto, in: Proc. of the International Conference on Deductive and Object-Oriented Databases (DOOD'89).
3. Booch, G., Rumbaugh, J. and Jacobson, I. (1999), The Unified Modelling Language User Guide. Addison-Wesley.
4. Firesmith, D. (1995), The Inheritance of State Models. Report on Object Analysis and Design, March-April, pag.13-15.
5. Maier, D. and Zdonik, S. (1989), Fundamentals of object-oriented databases, in: Readings in Object Oriented Database Management Systems. Maier and Zdonik, Morgan Kauffman.
6. Marcos, E. (2001). Defining taxonomic hierarchies: their implications for multiple inheritance. In Proceedings of the 13th. International Conference on Software Engineering & Knowledge Engineering (SEKE'2001). Knowledge Systems Institute, Sokokie, IL (USA), pag. 336-340.
7. Maughan, G. and Durnota, B. (1994), MON: An Object Relationship Model Incorporating Roles, Classification, Publicity and Assertions, in: Proc. of the International Conference on Object Oriented Information Systems (OOIS'94). Ed. by Patel, Sun and Patel, Spring Verlag, pag. 166-180.
8. Meseguer, J. (1993), Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming, in: Proc. of the European Conference on Object Oriented Programming (ECOOP'93), Springer Verlag.
9. Pirotte, A., Zimányi, E., Massart, D. and Yakusheva, T. (1994), Materialization: a powerful and ubiquitous abstraction pattern, in: Proc. of the International Conference on Very Large Databases (VLDB'94). Ed. by Bocca, Jarke and Zaniolo. Morgan Kaufmann, pag. 630-641.
10. Sloman, S.A. (1998), Categorical Inference Is Not a Tree: The Myth of Inheritance Hierarchies, Cognitive Psychology 35, No. 1, pag. 1-33.
11. Snoeck, M. and Dedene, G. (1996), Generalization/ speciali-zation and role in object oriented conceptual modeling. Data & Knowledge Engineering 19, pag. 171-195.
12. Taivalsaari, A. (1996), On the Notion of Inheritance. ACM Computing Surveys, Vol.28, No.3, Pag. 438-479.
13. Wang, Y., Storey, V.C., and Weber, R. (1999), An Ontological Analysis of the Relationship Constructor in Conceptual Modeling. ACM TODS 24(4), pag. 494-528.
14. Wieringa, R., De Jonge, W. and Spruit, P. (1995), Using Dynamic Classes and Role Classes to Model Object Migration. Theory and Practice of Software Systems, Vol.1(1), pag. 61-83.

Specialization/Generalization in Object-Oriented Analysis: Strengthening and Multiple Partitioning

Pieter Bekaert*, Geert Delanote, Frank Devos, and Eric Steegmans

Department of Computer Science
Katholieke Universiteit Leuven, Belgium
{pieter.bekaert,geert.delanote,frank.devos,
eric.steegmans}@cs.kuleuven.ac.be

Abstract. The driving force in object-oriented analysis to use the concept of specialization/generalization is polymorphism: the capability and need to reason about the union of the sets of objects of the specialization classes. Hereby features will be defined at the appropriate place. The fact that classes have common features is not a sufficient condition to generalize. The principle of strengthening specifications of features is an indispensable rule to manage class hierarchies. From the viewpoint of polymorphism, multiple specialization/generalization is only a logical extension of this modeling concept and not an optional or exotic one. Further, we discuss some guidelines to build sound class hierarchies, such as using (multiple) partitions.

1 Introduction

The major purpose of object-oriented (OO) analysis is to establish the requirements for the software system to be developed. In view of the OO paradigm, a conceptual model of relevant objects is developed including a specification of their characteristics, constraints and behavior [3,5]. This model describes the problem context in all its facets. The model will therefore be as close as possible to the facts, as they are observed in the external world.

Object-oriented analysis is a relative young research area. Many conceptual models are already oriented towards system implementation and a lot of design decisions are already taken. This can partly be explained by the fact that conceptual modeling draws part of its concepts from OO programming. The border between OO analysis and design is often vague. We are in favor of a strict separation between both activities. Every decision made by an analyst on *how* to represent an observed fact in a conceptual model should have been postponed to the design phase. An analyst has to concentrate on the reality and the requirements itself. A good OO analysis method

* Research Assistant of the Fund for Scientific Research □ Flanders (Belgium) (FWO □ Vlaanderen)

J.-M. Bruel and Z. Bellahs□ne (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 34-43, 2002.

has investigated and evaluated different alternatives. It has to impose rules on the concepts the method offers. This should lead to better, more extensible and reusable conceptual models.

During OO design we transform the conceptual model into an operational model. Software quality factors and other non-functional requirements will dominate the transformation. In the design phase we describe how a conceptual model can be implemented in terms of software. Other rules and guidelines apply here.

One of the key concepts in object-orientation is *specialization/generalization* (spec/gen). In this paper we investigate the use of spec/gen in OO analysis and we propose some methodological guidelines indicating when and how to use spec/gen. Section 2 describes the relationship spec/gen introduces between classes. In Sect. 3 we emphasize on polymorphic consideration as the guiding motivation to introduce spec/gen. Section 4 elaborates the rule of strengthening and gives some very powerful uses. In Sect. 5 we advocate the use of partitions, i.e. to use complete and disjoint specializations. In Sect. 6 this is extended to multiple spec/gen. The discussion is inspired by the EROOS method [15,13]. While EROOS offers its own notation, which is tailored to the concepts and principles of the method, we elaborate this paper using the *Unified Modeling Language* (UML) [12].

2 The Specialization/Generalization Relation

Specialization/generalization allows a class to be defined as a specialization of another more general class. In this paper the former is called the *specialization class* and the latter the *generalization class*.

In the semantics for a conceptual model, for each class a *population* is considered. At each point in time relevant in the reality that is modeled, the population of a class models the set of objects that exist at that time. The primary characteristic of the spec/gen relationship between two classes is that it introduces a *set inclusion* between the population of the specialization class and the population of the generalization class.

The set inclusion relation between a specialization class and the generalization class implies the very powerful possibility to use objects of specialization classes in a *polymorphic* way: everywhere where objects of the generalization class are considered, objects of any specialization class can be present in the reality/population.

An important related consequence is that all features are *inherited*. All attributes and associations present for objects of the generalization class are also present for objects of the specialization class and all constraints on the objects of the generalization class must equally apply for objects of the specialization class. All operations that are applicable to objects of the generalization class are equally applicable to objects of the specialization class.

Features can be introduced at different levels in the spec/gen hierarchy and *redefinition* of features at more specialized levels is possible. The question is then raised which definition applies when a feature is accessed through a polymorphic reference to an object. In OO programming, the dynamic binding process selects the most specific implementation for the operation that is applied. In OO analysis a comparable approach is used, with one difference: not only the most specialized

definition applies, but all definitions present in the concrete class of the object as well as in all direct and indirect generalizations apply. Some programming languages (e.g. Beta [6]) apply a comparable approach, referred to as refinement or Scandinavian semantics for methods. Due to procedural semantics this does not guarantee the conservation of semantics. In EROOS conceptual modeling however we advocate the use of declarative definitions for operations and logical conjunction is used to combine all definitions in the branch.

Thus, all knowledge specified and derived at the level of a certain class applies for all its objects, a fortiori for the objects of specialization classes. This realizes the (state as well as behavioral) subtype relationship and ensures compliance with the Substitution Principle of Liskov [7,8] that states that wherever an instance of a class is expected, one can always substitute an instance of any of its specialization classes. Thus in a part of the conceptual model that uses a certain generalization class, reasoning can be done based on the definition of this class and its own generalization classes, and will be valid, independent of the actual (and future) specializations.

Spec/gen will always be semantically motivated. We think that in analysis there is no place for inheritance that does not fulfill the subtype relationship. The EROOS method only allows building subtype hierarchies. OMT [11] allows the modeler to specify whether or not a given inheritance relation should be conformant. Other kinds and uses of inheritance are identified and classified in OO design and programming [9,4], some of which compliant with our approach, e.g. subclassing for specialization, subclassing for extension.

3 When to Generalize: Polymorphic Consideration

Specialization/generalization is a concept that combines two aspects: substitutability (involving polymorphism) and inheritance (with a focus on reuse). Both are used as motivation to define a class as a specialization class of another class. While reuse has been heavily focused upon, it is not a primary reason anymore.

We believe that in OO analysis the need for *polymorphic consideration* is the guiding principle: if the objects of different classes are not considered by the modeler as all being somehow instances of the same concept, a common generalization is not appropriate. The other way round, if classes are not related somehow in a spec/gen hierarchy, polymorphic consideration is in general not possible.

Polymorphic consideration is typically present in the model itself in the form of an association involving at one end objects that can come from more than one class. This urges for the definition of a common generalization class, which will participate in the association. Polymorphic consideration can also manifest itself in (formal) reasoning about possible populations and population evolutions. A large part of this reasoning will be done based on the definition of generalization classes, e.g. the class involved in the association. The principle of polymorphic consideration implies that spec/gen is a semantic relation and will lead to the semantic foundation for the generalization class. Therefore compliance with the substitution principle is of key importance.

Generalization merely based on reusing common features is ruled out. At the level of design and implementation, this use of inheritance might have its place. In analysis,

a `Person` and a `Product` will not be generalized only because they share a name with manipulation operations. Other concepts should be offered to enable reuse here.

However, when a motivation for spec/gen is found as described above, it is very important to generalize the features (and constraints) shared in all specialization classes. Hereby shared semantics of the feature is a prerequisite.

4 When and How to Specialize: Strengthening

Often only part of the semantics can be generalized. When differentiation is needed in the definition of certain features between different groups of objects of a class, specialization is needed, together with a refinement of the definition. As advocated above, this redefinition is limited: the objects of the specialization class must comply with all the definitions in the generalization classes and thus definitions in specialization classes should not lead to contradictions. The process of refinement or redefinition that obeys this rule is called *strengthening*.

4.1 Strengthening Structure

A first strengthening is the addition of extra constraints. Care has to be taken that these extra constraints do not give rise to the violation of the substitution principle for the operations. The specification approach for behavior in EROOS warrants this, thanks to the absence of preconditions, and the possibility of failure[1]. However this disallows the modeler to reason about success and failure in a polymorphic way.

Another possible strengthening is the strengthening of associations. We have the impression that this is used rather rarely and considered exotic. It has nevertheless a high potential and can be used often. In [12] an example is given involving paired class hierarchies between which an association is present and pair wise specialized. Due to the same possibility of failure, such a covariant redefinition poses no problems with respect to substitutability. Other examples that can often be used are illustrated in Fig. 1, using the notation of [12]. Central to these strengthenings is the preservation of the existential dependences. The existential dependence of a `Rent` from a `Person` is strengthened at the level of `Membership Rent` via an intermediate object of the `Membership` class. At the level of `Reservation Rent` this dependence, as well as the inherited dependence from a `Book` is strengthened via a single intermediate `Reservation` object. In certain configurations, the redefinition must be enriched with extra information (i.e. query redefinitions) to clear up ambiguities.

The rule about the preservation of existential dependences can be generalized for multiplicities: they can only be strengthened to a subset of the original. The strengthenings proposed here also preserve the intent and subset the extent of the parent. In EROOS this interesting use of strengthening is less confusing, due to reification of all associations, the central role of existential dependence and the implicit notation for strengthening associations.

---

[1] In EROOS, the postcondition is only guaranteed in case of success. In case of failure, the operation has no effect besides failing.

An alternative for some of the strengthenings of associations is the introduction of explicit constraints. We prefer however the use of visual implicit constraints [16] as indicated.

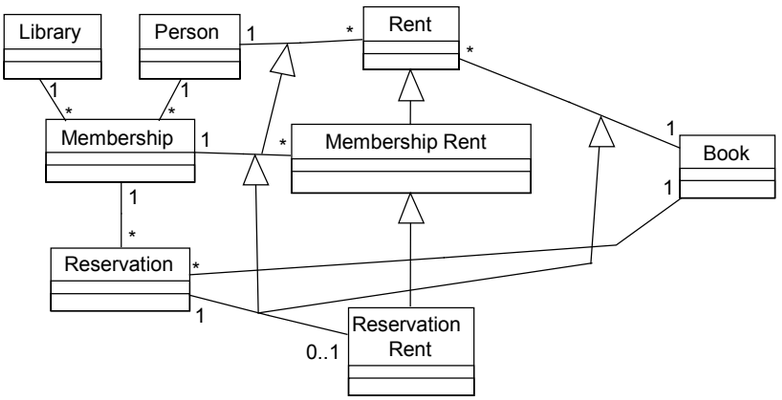


Fig.1. Strengthening associations (the relation between books and libraries has been left out)

4.2 Strengthening Behavior

The definition of operations can also be strengthened. This might e.g. be needed due to structural strengthening. The strengthening of an association might for instance require strengthening operations that manipulate that association.

Non-deterministic specification of behavior is a very powerful concept to separate the specification of structure from the specification of behavior [1,2]. Non-deterministic specifications can be strengthened to less non-deterministic ones.

5 Partitions

In this section we will investigate on organizing specialization/generalization hierarchies. Spec/gen introduces a hierarchical taxonomy, where the commonalities are specified in more general classes while the specifics are specified in more specialized classes. A specialization class is then not only to be considered in relation with its generalization class, but also with its siblings.

In many methods constructs are available to group specialization classes of a given generalization class and to put restrictions on these groups, although their use is not always stressed. Two constraints that can be imposed on top of such a group of specializations are *completeness* and *disjointness*. While completeness introduces extra semantics, disjointness[2] only restricts which hierarchies are correct.

[2] Assuming single instantiation.

If a group of specializations is restricted to be both disjoint and complete, they form a *partition* of the generalization class. EROOS imposes the use of partitions for structuring spec/gen hierarchies. One of the reasons is that we believe it leads to spec/gen hierarchies that are more stable with respect to later needs and changes in the problem domain. In Sect. 6.2, we will motivate that the partitioning principle, in combination with multiple specialization, can lead to better spec/gen hierarchies. The possible need for (polymorphic) use of a certain group of objects, as well as the need to introduce a feature for this group will be a driving force for partitions here.

Our partitioning principle is fulfilled in subtype inheritance as presented by Meyer in his taxonomy of taxonomies [10,9]. We do not reject the other forms of model inheritance that Meyer describes, but believe that also there the partitioning principle can and should be used.

The partitioning principle could also be interpreted as obstructing future evolution of the conceptual model. More in particular completeness might be seen as prohibiting the addition of extra specializations in a given partition when they appear in reality. This is not the case, since it is completely normal that the model changes when the (perception and conception) of the problem domain changes. The partitioning rule, together with multiple spec/gen using orthogonal partitions (see Sect. 6.2), even supports this change, since it helps to build hierarchies that are independent of the order in which they were created during the evolution of the problem domain. Moreover, polymorphic use and reasoning will support this addition without dramatic changes to the rest of the model.

6 Multiple Specialization/Generalization

Multiple specialization/generalization occurs when a specialization class is a specialization of more than one generalization class. This introduces with each generalization class in isolation a relationship as described in Sect. 2.

The population of the specialization class is a subset of the population of each of the generalization classes and thus of the intersection of the populations of all its generalization classes.

Each object of the specialization class inherits all features of all (direct and indirect) generalization classes. Hereby features with common names will automatically be unified.[3] For associations, role names are considered (however explicit multiple specialization of relations can also be used to indicate unification).

In OO programming multiple inheritance is often considered problematic among others due to the problems related with repeated inheritance and with need for strategies to decide which feature definition is selected in certain languages. In OO analysis however no such choices need to be made. *All* definitions for a given feature apply (thus also those found in all generalizations) as with single inheritance.

---

[3] The naming of features in the model should be done in a way that supports this. The possibility to rename features exists in certain approaches to resolve problems. This should however not be used to diverge features that were introduced as a single feature in a common generalization class in the hierarchy.

If clear, the combination of all definitions is implicitly assumed. It is also possible to further strengthen this definition. In any case, the rules of strengthening must apply for the spec/gen relation between the specialization class and each generalization class in isolation. If this is not realizable, the organization of the spec/gen relations and/or feature definitions is not appropriate.

6.1 Why Do We Need Multiple Specialization/Generalization?

In Sect. 3, polymorphic consideration is put forward as the guiding principle necessitating spec/gen. Certain needs for polymorphic consideration cannot be fulfilled using single spec/gen. This is the case if these needs involve object sets that have a non-empty intersection but are not subsets of one another. Multiple spec/gen is frequently present in the world and thus inherent in many problem domains. As a consequence, multiple spec/gen is not just a concept that can optionally be offered by a conceptual modeling method and that an analyst can choose to use or not.

Building a multiple spec/gen hierarchy[4] enables the definition and strengthening of features at the right place. The form of the spec/gen hierarchy will be compatible, and inspired by, the distribution of features over the different classes.

6.2 Building Good Multiple Specialization/Generalization Hierarchies

A good OO conceptual modeling method must provide rules and guidelines that lead to a good *organization* of the specialization/generalization hierarchy. The principle of polymorphic consideration is a good start. But other criteria are also important. First of all, the organization must also support abstraction, must comply with intuition and allow modular reasoning by separating concerns. Second, the organization must support the evolution of the problem domain. During the process of software development the perception and conception of the problem domain typically evolve. This implies changes to the spec/gen organization. The modeling method should lead to an organization that supports these changes with the least impact possible.

Rooted Class Hierarchies. In what follows, rooted class hierarchies are assumed. We believe that multiple spec/gen hierarchies should have roots, leading to diamond-shaped hierarchies. The reason is semantic: it is not appropriate in analysis to combine through multiple spec/gen two generalization classes that have no common semantics at all. These common semantics should be generalized.

Multiple Partitions. EROOS uses the partitioning principle to organize spec/gen. As a consequence of this principle, multiple spec/gen implies multiple partitions of a common generalization class and vice versa. This principle leads to partitioning the population of a generalization class along different dimensions according to orthogonal criteria. This is referred to as *orthogonal partitioning*.

Without further multiple specialization, no objects of an orthogonally partitioned class can exist, since an object can only directly belong to one concrete class. A concrete class that is an indirect specialization of an orthogonally partitioned class,

[4] Actually, it is of course not a hierarchy in strict sense.

has to be a specialization of exactly one class from each partition: at least one due to completeness and at most one due to disjointness. Every combination of classes of the different partitions that is possible in the real world must be introduced explicitly in the conceptual model. Thus it becomes possible to rule out certain combinations.

In a deeper multilevel hierarchy, care must be taken not to violate this rule. Not selecting classes from each partition for instance makes the class abstract: further multiple specialization below in the hierarchy will still be needed.

Orthogonal partitioning stimulates the separation of concerns. It allows other parts of the conceptual model to consider only one (or some) of the partitions. This separation has already been advocated before as a good decision (e.g. [11]), but we impose it via disjoint specialization, pushing the modeler to search for real orthogonal criteria (with a finer granularity), and thus to not mix different dimensions. Completeness ensures the full consideration of the criterion, by classifying all objects with this respect, making the generalization class abstract. An incomplete group of specializations for instance does not offer the possibility to specify (without extra constraints) the use of an object of the generalization class that does not belong to one of the specializations. Imposing completeness and disjointness from the beginning avoids changes related to already covered dimensions due to new needs for polymorphic use.

When a new criterion is discovered, adding an extra partition can be done without touching the existing partitions. New combinations must be made, but existing use of the hierarchy does not have to be changed.

Managing Complex Hierarchies. Suppose we want to build a good spec/gen hierarchy given the observation of sets of (potential) objects. Each set indicates polymorphic consideration and/or the observation of a semantically common feature (or strengthening thereof). These sets can be arbitrarily related to each other.

A full hierarchy can be built in which all features or strengthenings can be specified once at the appropriate place and in which all current needs for polymorphic consideration *as well as all possible future needs* based on combinations of current sets can be fulfilled without change. A possible hierarchy will have at the top level a binary partition for each observed set, and below a well-organized hierarchy of combinations. The future discovery of new sets can be merged with this hierarchy.

The anticipation of *all possible* future polymorphic needs leads however to an overly complex and exponentially large hierarchy. Moreover this does not exploit the intuitive nature of taxonomies in the observed reality.

One way to relax this complexity is to replace binary partitions with partitions involving more than two specializations (see Fig. 2). This can be done if such a partition is present (A-B-C) in the sets mentioned above and if the need for neither the complement (e.g. X\A) nor the union (e.g. A∪B) of some of the specialization classes is observed or expected in the future. Another way is based on the observation of subset relationships among the sets (see Fig. 3): the partition introduced by a certain (set of) set(s) (V) could be placed under a class corresponding with the superset (U). Also here, the criterion is the expectation that the complement (e.g. Y\V) of none of the sets, with respect to a larger set (Y) than the union (U), is not needed, i.e. that the semantic of the dimension introduced by the partition is local.

This bottom-up taxonomy experiment illustrates that in building a good spec/gen hierarchy a lot of common sense and realistic expectations about future needs are needed to manage the trade-off between simplicity and anticipation of future needs. Moreover good tool support can help us deal with the complexity and evolution.

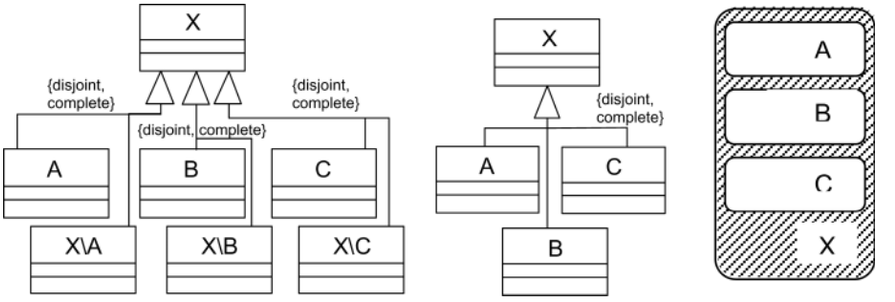


Fig.2. Introducing partitions with more than two specializations (further combining specializations left out)

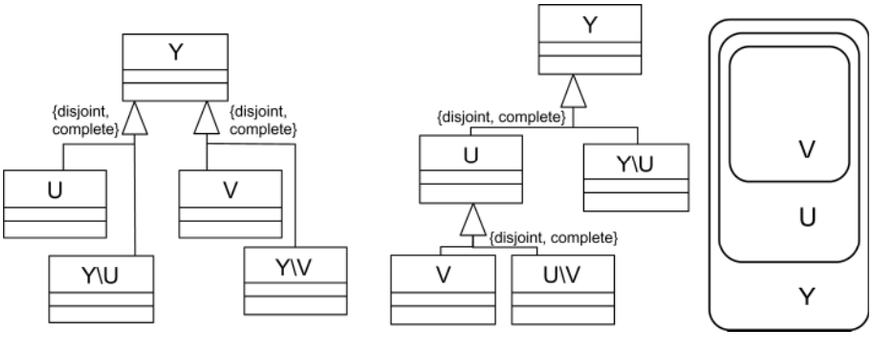


Fig.3. Moving partitions down in the hierarchy (further combining specializations left out)

7 Conclusion

In this paper, we presented our view on specialization/generalization in OO analysis. The principle of polymorphic consideration is used to decide when to introduce a generalization. Strengthening guides the redefinition of features at different levels. Partitioning is put forward as an instrument to build good and stable spec/gen hierarchies. The observations have been generalized for multiple spec/gen.

An interesting topic for further consideration is among others static versus dynamic classification or roles [14]. In EROOS currently classification is static, but we believe dynamic classification can be integrated in our approach, as an option for each partition. A related issue for further investigation is when to use alternatives of spec/gen, such as delegation (or reification of roles as another associated class) or classes modeling types. Also the trade-off described in the last section, as well as the

position of non-disjoint and incomplete groupings of classes instead of partitions in this context, asks for further research.

References

1. P. Bekaert and E. Steegmans. Using Non-Determinism for the Separate Specification of Structure and Behaviour. Report CW 337, Department of Computer Science, K.U.Leuven, Leuven, Belgium, April 2002.
2. P. Bekaert and E. Steegmans. Non-determinism in conceptual models. In Proceedings of the Tenth OOPSLA Workshop on Behavioral Semantics, OOPSLA'01, October 2001.
3. Borgida. Features of Languages for the Development of Information Systems at the Conceptual Level. IEEE Software, 2(1), January 1985, pp. 63-73.
4. T. Budd. An Introduction to Object-Oriented Programming. Addison Wesley, 1997.
5. S. Greenspan, J. Mylopoulos, and A. Borgida. On Formal Requirements Modeling Languages: RML Revisited. International Conference on Software Engineering, 1994, pp. 135-147.
6. Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. Object-Oriented Programming in the Beta Programming Language. Addison-Wesley, 1993.
7. Barbara Liskov. Data Abstraction and Hierarchy. SIGPLAN Notices, 23,5 (May, 1988).
8. Robert C. Martin. The Liskov Substitution Principle. C++ Report, March 1996.
9. B. Meyer. Object-Oriented Software Construction, 2nd edition. Prentice Hall, 1997.
10. B. Meyer. The Many Faces of Inheritance: A Taxonomy of Taxonomy. IEEE Computer, 29(5):105–108, May 1996.
11. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. Object-oriented modeling and design. Prentice-Hall Englewood Cliffs (N.J.), 1991.
12. J. Rumbaugh, L. Jacobson, and G. Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, 1999.
13. E. Steegmans, P. Bekaert, F. Devos, J. Dockx, B. Swennen, and S. Van Baelen. Object-Oriented Analysis with EROOS. Department of Computer Science, K.U.Leuven, Belgium, 2001.
14. F. Steimann. On the representation of roles in object-oriented and conceptual modeling. Data & Knowledge Engineering, 35, 2000, pp. 83-106.
15. S. Van Baelen, J. Lewi, E. Steegmans, and H. Van Riel. EROOS : An Entity-Relationship based Object-Oriented Specification Method. Technology of Object-Oriented Languages and Systems TOOLS 7 (B. G. Heeg and B. Meyer, eds.), Prentice-Hall, Hertsfordshire, UK, 1992, pp. 103-117.
16. S. Van Baelen, J. Lewi, and E. Steegmans. Constraints in object-oriented analysis and design. Proceedings of Technology of Object-Oriented Languages and Systems TOOLS 13 (B. B. Magnusson, ed.), 1994, pp. 185-199.

Towards a New Role Paradigm for Object-Oriented Modeling

Stéphane Coulondre[1] and Thérèse Libourel[2]

[1] LISI, Bât Blaise Pascal (501), INSA de Lyon
20 Av Albert Einstein, 69621 Villeurbanne Cedex, France
Stephane.Coulondre@insa-lyon.fr
[2] LIRMM
161 rue Ada, 34392 Montpellier Cedex 5, France
Libourel@lirmm.fr

1 Introduction

In this paper, we consider the framework of "complex" information systems, related to biology, geographical information, multimedia, etc. Apart from being inherently complex, objects of these domains are evolutive, and related applications require more and more often to take into account this functionality, in the same way that reusability and extensibility currently are.

What does it mean exactly to bring evolution in a formalism as a basic functionality ? It means proposing, from the very first step of analysis and modeling, a way to express that future objects will, on one hand be persistent, and on the other hand be able to evolve through their life. This can be realized by allowing them to dynamically gain or loose some pieces of structure or behaviour aspects.

This capability is practically added by taking into account the "role" notion as a complement of the notions of concept or class. In traditional class-based models, an object is linked to a single class (its generating abstract model), according to the mono-instanciation principle, and this class determines the object's structure and behaviour during its lifespan.

Indeed, some propositions, as that of P. Coad [6], suggested that objects could turn up to play several roles (cf. figure 1), but the role is considered as a concept located on the same level as the class.

A quick survey of object-modeling past work [14,13,12,10] reveals three essentials features of the evolution functionality :

- the ability for every object to change its structure and behaviour during its lifespan,
- the ability for every object to play several roles at the same time,
- the ability for every object to enter the systm with a given "base" role, then to dynamically gain or loose other roles, thus respecting the Aristotelician way of considering objects.

In this paper, we present an approach for extending the UML formalism, in order to take into account evolution of objects. The extension we propose is

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 44–52, 2002.

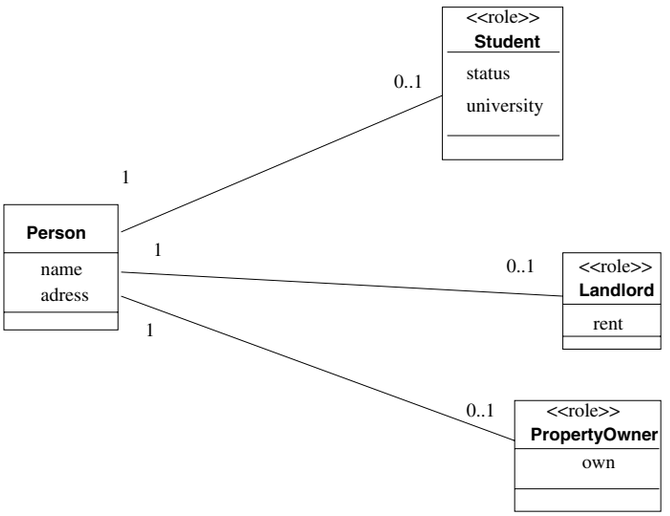


Fig. 1. Classe and roles

based on the main concepts of the database model called *Samovar*, which is the first object-role database model that is fully ODMG-compliant[4,1,5]. The goal of our current work is to provide a complete paradigm for evolution, from the modeling phase (which is the subject of this paper) to the implementation and querying phases (which is the subject of the Samovar database system).

The extension we propose is based on the features of UML 1.4 meta-model, in which the concepts of classifiers and associations are fundamental. We will emphasize in the conclusion that the various proposals for UML 2.0 consider the role concept as part of the model, but with a noticeable different meaning. Indeed, the goal is to consider modelling the whole system behaviour from the angle of roles.

To our knowledge, only little work has considered evolution in an homogeneous way, from modeling to implementation and querying, such that the ODMG standard prequesites are respected [9,3,11,15] . For a survey on role management within object-oriented languages and database systems, we refer the reader to [8,7].

The outline of the paper is as follows: section 2 introduces the basic concepts of the Samovar model. Section 3 then describes the proposed UML extensions for taking into account evolution in the modeling phase. Section 4 finally concludes the paper.

2 Basic Concepts

We shall introduce the core concepts of the Samovar model we rely on:

2.1 Types

Our model supports three kinds of types: atomic types, constructed types and user types. Atomic types are: integer, char, string, boolean, real and bits. Constructed types are obtained from atomic types by recursive application of the following constructs: tuple, set and list. Given a class name n, n is a user type, as well as $\%n$, which is the view type on the class n, related to the new notion of view on instances of n, presented later.

2.2 Classes and Roles

A class contains a set of roles. Each role is defined by a *criterion*, a type and a set of method signatures. Figure 2 identifies the roles in a conceptual manner, with simple terms, such as *Physical, Student, Chemical*, etc., reflecting the desired semantics, for reasons of clarity. Actually, criteria are not simple syntactic terms, but a restricted form of first-order logic (FOL) formulae *freely chosen* by the user, in order to reflect the desired semantics. For example, the class *Animal* has six roles: *Common, Physical, Chemical*, *Biological*, *Predator* and *Prey*, which can, for example, be respectively defined as *true, state(physical), state(chemical), context(biological), context(biological)* $\wedge$ *state(predator),* and *context(biological)* $\wedge$ *state(prey),* or with any other formula the user will choose.

Criteria semantics induce a partial order, defined by FOL logical implication. Figure 2 shows the role hierarchies within classes. For example, in the *Animal* class, as *context(biological)* $\wedge$ *state(predator)* implies *context(biological),*

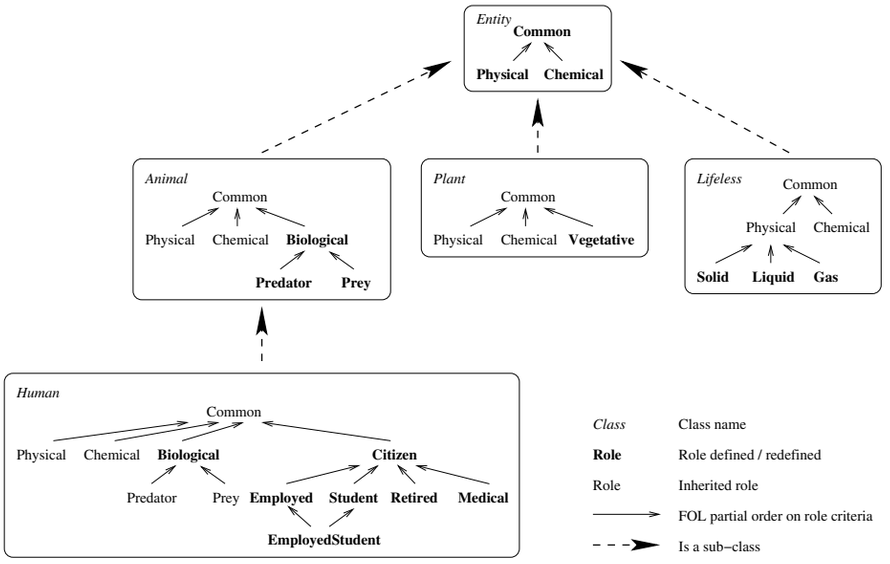


Fig. 2. Class and role hierarchies

then *Predator* is located under *Biological* in the role hierarchy. FOL provides a declarative approach to modeling, and this declarative paradigm provides interesting additional features such as property sharing between roles or declarative valuation, as illustrated in the next section.

Each of these roles has its own associated type and set of method signatures. Indeed, no type or method is directly associated to classes. A class is intuitively seen as a bag of role definitions. It only describes the set of types and operations that can be assumed by its instances - i.e., all the roles they can play.

Classes are themselves organized in another hierarchy, built on the *Is a sub-class* link, with possible multiple inheritance. Unlike in standard object models, no type is directly assigned to classes. Thus the *Is-a-sub-class* link is different here, entailing constraints on the form of the role hierarchies each class contains, and also on the types and methods of these roles. Therefore, a class inherits its role hierarchy from its super-classes, and can extend these by adding or refining roles types, as well as adding or redefining methods. Figure 2 illustrates a class hierarchy. The top is the *Entity* class. For example, the *Biological* role is defined in the *Animal* class, and redefined in its *Human* sub-class.

2.3 Objects

An object is an instance of a class. It can play one or more roles defined in its class, and gain or lose them dynamically. It always plays the top role of the role hierarchy defined in its class. It has a specific identifier (OID). The set of roles played by an object is always a sub-hierarchy of the role hierarchy defined in its class. To each of these played roles is assigned a value within the object that conforms to the corresponding type. Figure 3 shows the structure of the object *Peter* with an OID of 28. It plays only five roles, which are *Common*, *Physical*, *Chemical, Citizen* and *Student*. Values associated with each role conform to the corresponding types in the *Human* class.

2.4 Hierarchies

The hierarchies introduced above induce several consequences on object capabilities:

- Role hierarchies: if r' *is located under* r, then each object playing the role r' also plays the role r. Therefore, as *Peter* plays the *Student* role, it also plays the *Citizen* role. It therefore has the properties *surname* and *firstname*, and it can invoke the methods of *Citizen*.
- Class hierarchies: if o is an instance of the class c, and if c *is a sub-class* of c', then o can play all the roles defined in c'. Thus, as *Peter* is an instance of the *Human* class, it can play all the roles of the *Animal* class and of the *Entity* class.

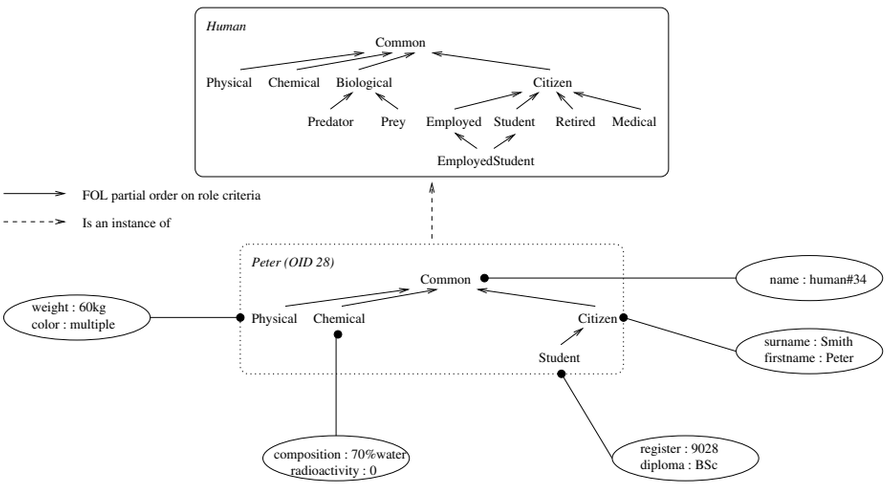


Fig. 3. A class instance

3 Potential Extension of the UML Formalism

The UML formalism [16,2] is presently used in numerous object-oriented analysis and modeling cases. One of its strength is to offer a communication channel through a projection of the analyzed system, this projection being obtained from graphical representation. Its strength relies on the definition of the meta-model in which all the elements necessary for the object-oriented approach are defined. However, one must note that some elements are still lacking, in order the formalism to be able to answer some natural situations in which the objects evolution must be taken into account.

Therefore, we propose to extend the notion of "role" within the UML formalism. The concept of role in UML is considered as "association role". Indeed, if we refer to the excerpt of UML meta-model, related to associations (cf. figure 4), classes that are implicated as *AssociationEnd* in an association can play a role in this association which can be described in the class diagram.

As presented in figure 5, our proposition is to insert, at the meta-model level, a specialization of *Classifier*.

Current *Classifiers* are elements that are specialized in a "static" way, according to the modeler wish (for example by creating a generalization-specialization link between classes). A role can be introduced as a "dynamic" classifier because it contains *StructuralFeature* and *BehavioralFeature* as traditional classifiers, but also *LogicalFeature* (or criterion) and its specialization is automatically deduced according to its criterion. The *Classifier* element is refined in *Class* (with a traditional meaning) and in *D_Class* (the new way to define classes in terms of an aggregation of roles).

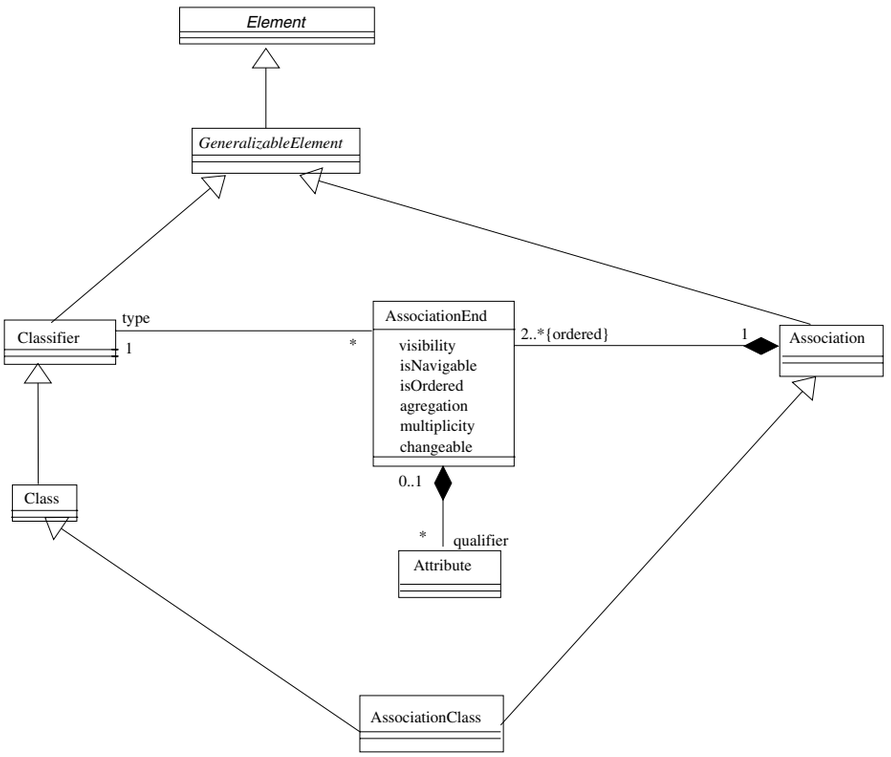


Fig. 4. UML meta-model association and classes

Moreover, the UML metamodel must be enriched by adding automatic specialization (*Is-an-extension* link between roles) and specialization-generalization between *D_class*(es) (*Is-a-subclass* link) to *RelationShip* elements.

Within a given model (cf. figure 6), we could therefore have the concept of *Dynamical_Classifier* as a stereotype role. The *D_class* would be constructed as a *Classifier*, but it would be here an aggregate of roles, not a set of attributes and operations.

Concerning the instanciation, the constructors associated to the *D_class* can specify which role is the base role.

4 Conclusion and Perspectives

The object-oriented paradigm is indeed largely used because of the powerful mechanism of specialization and generalization which allows the modeler to factorize the common characteristics, thus simplifying his/her tasks.

However, the principle of mono-instanciation remains a strong constraint which ties the modeler to a fixed vision of objects.

Introducing the role notion as we present it in this paper, adds a declarative part (criteria) to the existing formalism, and allows a dynamic classification. The role hierarchy is induces by the system by logical implication. The modeler can continue to refine classes as usual, because the traditional specialization-generalization link is still conserved.

It seems to us that these double-semantics of classification is underlying and necessary to the evolutive nature of objects we presented in the introduction.

The evolution of UML towards realease 2.0 extends the modelling process by considering two perspectives or viewpoints: Class Perspective (CP) and Role Perspective. It seems that our proposition lies in the CP perspective, for which we think the concept of classifier can be treated in a different manner. However, the RP perspective is a interesting approach from which Samovar may implement some additional features, especially as we study the opportunity of using it in the framework of software components.

As stated before, the proposition presented in this paper, has been implemented and used in the context of the Samovar database system. Our goal is thus to provide a full framework for evolution from modeling to implementation and querying.

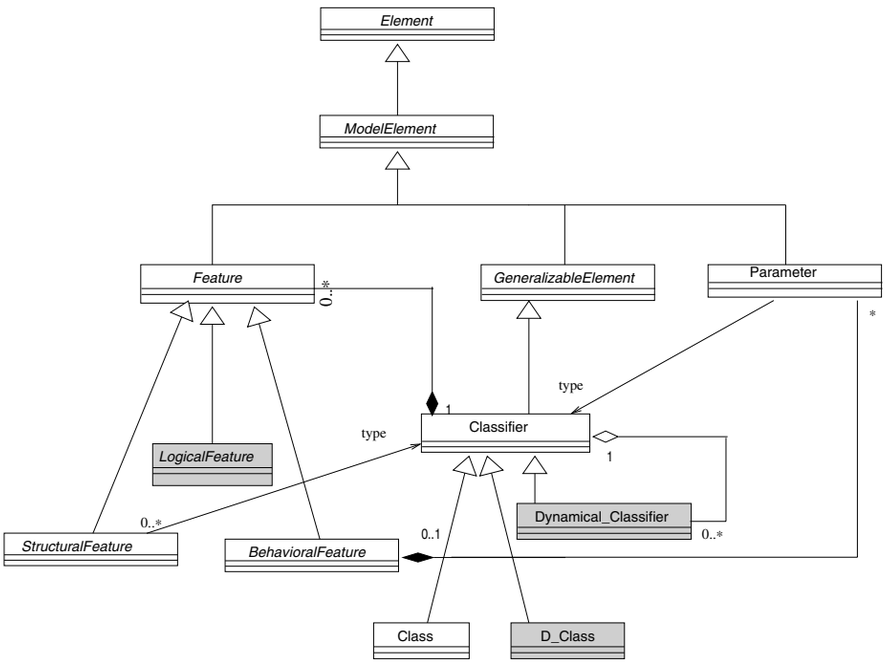


Fig. 5. Extension of UML meta-model

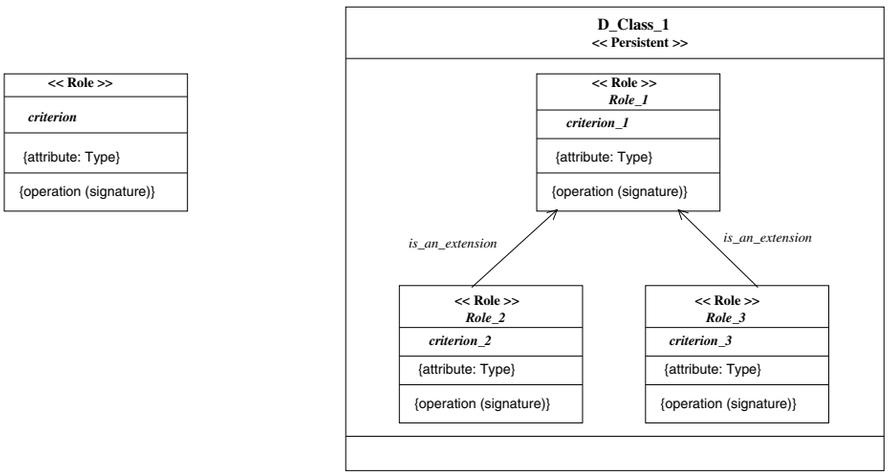


Fig. 6. Model with roles and *D_Class*(es)

References

1. T. Atwood, D. Barry, J. Dubl, J. Eastman, G. Ferran, D. Jordan, M. Loomis, and D. Wade. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1996. 45
2. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, USA, 1 edition, 1999. 48
3. H. Bowman, J. Derrick, P. Linington, and M. Steen. Cross viewpoint consistency in Open Distributed Processing. *Software Engineering Journal*, 11(1):44–57, January 1996. 45
4. R. G. G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Los Altos (CA), USA, 1997. 45
5. R. G. G. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann Publishers, Los Altos (CA), USA, 2000. 45
6. P. Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):153–159, September 1992. 44
7. S. Coulondre and Th. Libourel. The Samovar Database System. Technical report, LIRMM - Université Montpellier II / C. N. R. S., 2000. 45
8. S. Coulondre and Th. Libourel. An Integrated Object-Role Oriented Database Model. *Data and Knowledge Engineering*, 42(1):113–141, 2002. 45
9. W. H. Harrison, H. Kilov, H. L. Ossher, and I. Simmonds. Technical note — from dynamic supertypes to subjects: A natural way to specify and develop systems. *IBM Systems Journal*, 35(2):244–256, 1996. 45

10. Bent Bruun Kristensen and Kasper Østerbye. Roles: Conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems*, 2(3):143–160, 1996. 44
11. Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In Norman Meyrowitz, editor, *OOPSLA'89 Conference Proceedings: Object-Oriented Programming: Systems, Languages, and Applications*, pages 397–406. ACM Press, 1989. 45
12. J. Martin and J. Odell. *Object-Oriented Analysis & Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992. 44
13. M. P. Papazoglou. A methodology for representing multifaced objects. In *Proceedings of International Conference on Database and Expert Systems Applications (DEXA'91), Berlin, Germany*, 1991. 44
14. Barbara Pernici. Objects with roles. In *Proceedings of the Conference on Office Automation Systems*, Organizational Data Models, pages 205–215, 1990. 44
15. Trygve Reenskaug. *Working with Objects: The OOram Software Engineering Method*. Manning Publications, 1996. 45
16. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, Massachusetts, USA, 1 edition, 1999. 48

Analysing Object-Oriented Application Frameworks Using Concept Analysis

Gabriela Arévalo[2] and Tom Mens[1]

[1] Postdoctoral Fellow of the Fund for Scientific Research – Flanders
Programming Technology Lab, Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
tom.mens@vub.ac.be
[2] Software Composition Group, University of Berne
Bern, Switzerland
arevalo@iam.unibe.ch

Abstract. This paper proposes to use the formal technique of *Concept Analysis* to analyse how methods and classes in an object-oriented inheritance hierarchy are coupled by means of the *inheritance* and *interfaces* relationships. Especially for large and complex inheritance hierarchies, we believe that a formal analysis of how behaviour is reused can provide insight in how the hierarchy was built and the different relationships among the classes. To perform this analysis, we use behavioural information provided by the *self sends* and *super sends* made in each class of the hierarchy. The proposed technique allows us to identify weak spots in the inheritance hierarchy that may be improved, and to serve as guidelines for extending or customising an object-oriented application framework. As a first step, this paper reports on an initial experiment with the *Magnitude* hierarchy in the *Smalltalk* programming language.

1 Introduction

Understanding a software application implies to know how the different entities are related. In the case of an object-oriented application framework, our entities are classes and methods. When a developer defines a class in an application, he requires knowledge about how behaviour and structure have to be reused using inheritance techniques. It is not trivial to achieve optimal reuse, especially when the number of classes is large or the inheritance hierarchy is deep. In these situations, *concept analysis* can be used as a technique to help us cope with these problems, by analysing the inheritance and interface relationships among the classes in the class hierarchy. Then we can understand and document the way inheritance is used in the framework, and use this information to provide guidelines for how the framework can be modified or customised without running into behavioural problems or without breaching the design conventions used when building the framework.

Concept Analysis (CA) is a branch of lattice theory that allows us to identify meaningful groupings of *elements* (referred to as *objects* in CA literature) that

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 53–63, 2002.

have common *properties* (referred to as *attributes* in CA literature) [1]. These groupings are called *concepts* and capture similarities among a set of *elements* based on their common *properties*. Mathematically, concepts are *maximal collections of elements sharing common properties*. They form a complete partial order, called a *concept lattice*, which represents the relationships between all the concepts [1,15,6]. To use the CA technique, one only needs to specify the properties of interest on each element, and does not need to think about all possible combination of these properties, since these groupings are made automatically by the CA algorithm.

2 Applying Concept Analysis to Inheritance Hierarchies

In this paper, we report on an experiment that uses concept analysis to analyse an existing inheritance hierarchy with the aim to better understand how inheritance is used in practice to achieve reuse, and to provide guidelines to improve the inheritance hierarchy. To achieve this, we analyse classes and their methods based on their relationships in terms of *inheritance*, *interfaces* and *message sending behaviour*. The *inheritance relationship* indicates whether a class is an ancestor or descendant of another one. The *interface relationship* indicates which methods are defined abstract or concrete in each class. The *message sending behaviour* indicates which methods are called by other methods in a class. Because we are mainly interested in reuse of behaviour, we will only look at *self sends* and *super sends*.

As a first step, we need to define the *elements* and *properties* we wish to reason about to apply the CA technique. Because we are interested in classes in an object-oriented hierarchy, together with their methods and the messages sent by these methods, we define an *element* as a pair (C, s) such that "a method with signature s is called (via a self send or super send) by some method implemented in the class C". For the CA properties, we chose a classification based on the relationships explained previously:

Classification based on message sending behaviour. (C, s) satisfies predicate ***calledViaSelf*** if s is called via a self send by some method in C. (C, s) satisfies predicate ***calledViaSuper*** if s is called via a super send by some method in C.

Classification based on interface relationship. (C, s) satisfies predicate ***isConcreteIn:*** D if s is implemented as a concrete method in class D. (C, s) satisfies predicate ***isAbstractIn:*** D if s is implemented as an abstract method in class D.

Classification based on inheritance relationship (C, s) satisfies predicate ***isDefinedInAncestor:*** D if D defines s and is an ancestor class (i.e., a direct or indirect superclass) of C. (C, s) satisfies predicate ***isDefinedInDescendant:*** D if D defines s and is a descendant class (i.e., a direct or indirect subclass) of C.

[1] We prefer to use the terms *element* and *property* instead of *object* and *attribute* because the latter terms have a specific meaning in the object-oriented paradigm.

(C, s) satisfies predicate ***isDefinedLocally*** if C defines s. This means that s is defined in the same class that calls it.

CA properties are then defined as conjunctions obtained by taking one predicate from each classification. Below, we present some of the properties that can be obtained by a conjunction of the predicates presented previously.

- Predicate ***concreteSuperCaptureIn:*** D is a conjunction of *calledViaSuper*, *isConcreteIn:* D and *isDefinedInAncestor:* D. (C, s) satisfies this predicate if s is called via a super send in some method of C, and the receiver method is implemented in the class D that is an ancestor class of C.
- Predicate ***concreteSelfCaptureLocally:*** C is a conjunction of *calledViaSelf*, *isConcreteIn:* C and *isDefinedLocally*. (C, s) satisfies this predicate if s is called via a self send in some method of C, and the receiver method is defined as a concrete one in the same class C.
- Predicate ***concreteSelfCaptureInAncestor:*** D is a conjunction of *calledViaSelf*, *isConcreteIn:* D and *isDefinedInAncestor:* D. (C, s) satisfies this predicate if s is called via a self send in some method of C, and the receiver method is defined as a concrete one in the class D that is an ancestor class of C.
- Predicate ***concreteSelfCaptureInDescendant:*** D is a conjunction of *calledViaSelf*, *isConcreteIn:* D and *isDefinedInDescendant:* D. (C, s) satisfies this predicate if s is called via a self send in some method of C, and the receiver method is defined as a concrete one in the class D that is a descendant class of C.
- Predicate ***abstractSelfCaptureLocally:*** C is a conjunction of *calledViaSelf*, *isAbstractIn:* C and *isDefinedLocally*. (C, s) satisfies this predicate if s is called via a self send in some method of C, and the receiver method is defined as an abstract one in the same class C.

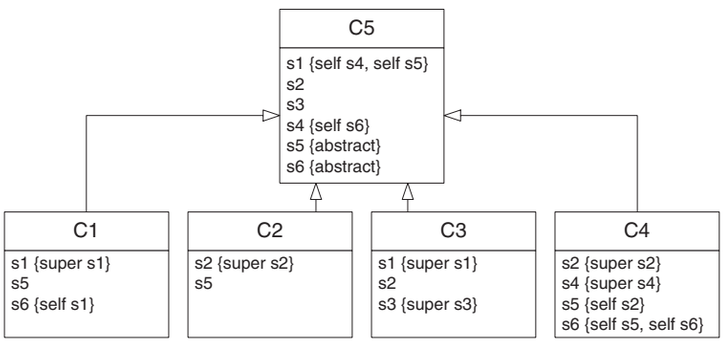


Fig. 1. Example class hierarchy

As an example of how these properties can be used to compute *concepts*, take a look at the example class hierarchy of Figure 1. All self sends and super sends in the source code have been annotated between curly braces. Based on this information, the CA algorithm will automatically compute the following concepts (among others):

- **Concept 1** has elements { (C_4, s_2), (C_4, s_5), (C_4, s_6) } and properties { *concreteSelfCaptureLocally:* C_4 }. This means that only the selectors s_2, s_5 and s_6 are called via a *self* send that is captured by *concrete* method implementations in the class C_4 itself.
- **Concept 2** has elements { (C_1, s_1), (C_2, s_2), (C_3, s_1), (C_3, s_3), (C_4, s_2), (C_4, s_4) } and properties { *concreteSuperCaptureIn:* C_5 }. This means that only the selectors s_1, s_2, s_3 and s_4 are called via a *super* send in the classes C_1, C_2, C_3, C_4 and they are implemented by a *concrete* method in the ancestor C_5 of these classes.
- **Concept 3** has elements { *(*C_5, s_5*), (*C_5, s_6*)* } and properties { *abstractSelfCaptureLocally:* C_5*, concreteSelfCaptureInDescendant:* C_1*, concreteSelfCaptureInDescendant:* C_4*,* }. This means that abstract methods s_5 and s_6 in C_5 are defined concrete in the subclasses C_1 and C_4.

In the remainder of this paper, we will make the concept notation more compact by grouping together all selectors that belong to the same class. For example, the element set of **concept 2** can be abbreviated to {(C_1,s_1), (C_2,s_2), (C_3,{s_1, s_3}), (C_4,{s_2, s_4})}. We will also identify each concept by a unique number that is automatically assigned to the concept by the CA algorithm.

3 Case Study

The abstract example of section 2 was only intended to make the reader understand how the process works. Our actual experiment consists of applying the CA technique to study the *Magnitude* inheritance hierarchy of Smalltalk in more detail[2]. We decided to use the *Magnitude* hierarchy for our first experiment because: it is sufficiently large to get meaningful results (29 classes, 894 methods); it heavily relies on code reuse by inheritance (19 abstract methods, 296 self sends, 49 super sends); it is stable and well-documented; it is commonly available for most versions of Smalltalk. Figure 2 displays the part of the *Magnitude* hierarchy that is used for the examples later in this paper.

Based on results provided by the CA algorithm, we analyzed the relationships between the classes in terms of *inheritance*, *interface* and *message sending behaviour*. With SOUL, a logic meta-programming language built on top of – and tightly integrated with– Smalltalk [17], we extracted 248 elements and 73

[2] For our experiments, we worked in VisualWorks release 5i4, and restricted ourselves to only those classes belonging to the Smalltalk namespaces Core, Graphics, Kernel, and UI.

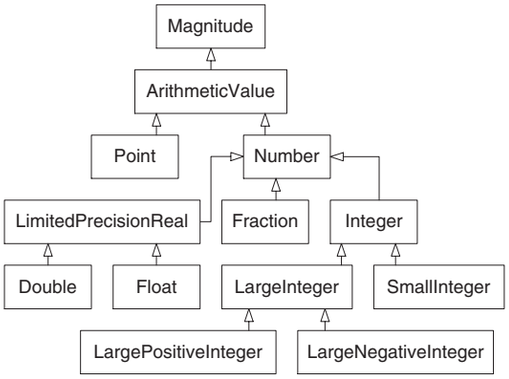


Fig. 2. Smalltalk *Magnitude* class hierarchy

properties. [3] Based on this, the CA algorithm that we implemented directly in Smalltalk computed 125 concepts as a result.

As we said previously, the properties will be of the form *concreteSuperCaptureIn: C, concreteSelfCaptureInDescendant: C, abstractSelfCaptureLocally: C, concreteSelfCaptureLocally: C, concreteSelfCaptureInAncestor: C, ...*. If we abstract the argument C out of these properties, we find that many concepts resemble each other because they contain the same set of properties. This commonality between concepts allows us to identify *concept patterns*. A *concept pattern* consists of a textual and graphical description, a concrete example related to the *Magnitude* class hierarchy, and an analysis of how the pattern provides more insight in how parts of the code are reused.

Concept Pattern 1: Self Sends Captured Locally

A set of selectors $m_1 \ldots m_p$ are called via a *self* send in a class B and they are implemented in the same class. Figure 3 shows this concept pattern graphically. It occurs in 21 concepts of the *Magnitude* concept lattice.

For example, **concept 71** has elements {*(Fraction,* {*reduced, negative, asFloat, asDouble*}*)*} and properties {*concreteSelfCaptureLocally: Fraction*}.

This concept pattern is useful to document the *internal interface* of a class, i.e., the set of all selectors that are implemented in the class and to which self sends are made by methods implemented in the same class. This internal interface captures and documents the core behaviour of the class. This can be used to distinguish the core methods of a class from the auxiliary ones. This is important information for reusers because, if the core methods are overridden in a subclass, all auxiliary methods will still work correctly with the new core [10].

[3] For our experiment we computed a static approximation of the self sends and super sends. For example, even if sends occur in a conditional branch that is never executed, they are still extracted by our algorithm.

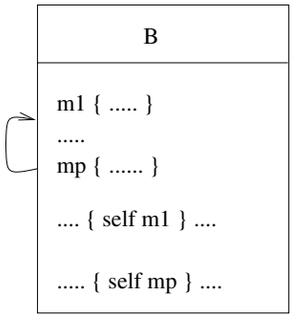


Fig. 3. Concept pattern 1

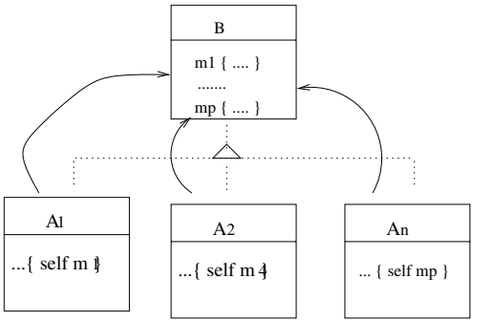


Fig. 4. Concept pattern 2

Concept Pattern 2: Self Sends Captured in Ancestor

A set of selectors $m_1 \ \ldots \ m_p$ implemented in a class B are called via *self* send in descendant classes $A_1 \ \ldots \ A_n$. Figure 4 displays this concept pattern graphically. It occurs in 9 concepts of the *Magnitude* concept lattice.

For example, **concept 73** has elements {*(LargePositiveInteger,* {*digitLength, digitAt:*}*), (LargeNegativeInteger,* {*digitLength, digitAt:*}*)*} and properties {*concreteSelfCaptureInAncestor: LargeInteger*}.

This concept pattern is useful to detect the *actual subclass interface* of a class, i.e., the set of all selectors that are implemented in the class and to which self sends are made by subclasses. Changes to these methods will also have an impact in all subclasses that reuse it. For example, using **concept 73** we know that if we change the implementation of *digitLength* or *digitAt:* in *LargeInteger*, we have to check whether the methods in *LargePositiveInteger* and *LargeNegativeInteger* that call these methods still behave as expected.

In terms of software refactoring [4], the concept pattern can sometimes be used to identify common code in sibling classes that is useful to refactor in the common superclass. For example, **concept 73** illustrates that, to a cer-

tain extent, sibling classes *LargePositiveInteger* and *LargeNegativeInteger* reuse the behaviour defined in their superclass *LargeInteger* in the same way. Further investigation of the actual source code allows us to discover that the self sends (digitLength: and digitAt:) are invoked from within the implementation of the method *compressed* in both sibling classes and the implementation of this method is very similar in both cases. Hence, a refactoring might be appropriate to extract this common behaviour into an auxiliary method that can be pulled up into the common superclass *LargeInteger*. This analysis showed us a limitation of our approach: we should not only take the receiver of a self send into account (in this case *digitLength* and *digitAt:*) but also the sender (in this case *compressed*), since this represents essential information.

Concept Pattern 3: Super Call

A set of selectors $m_1 \ldots m_p$ implemented in the class B are called via a *super* send in descendant classes $A_1 \ldots A_n$. Figure 5 illustrates this concept pattern graphically. It occurs in 8 concepts of the *Magnitude* concept lattice.

For example, **concept 105** has elements {*(Float,* {$>$, $\geq$, $\leq$}*), (Double,* {$>$, $\geq$, $\leq$}*), (SmallInteger,* {$>$, $\geq$, $\leq$}*), (LargeInteger,* {$>$, $\geq$, $\leq$}*)*} and properties {*concreteSuper-CaptureIn: Magnitude*}.

This concept pattern can be used to detect the *actual overriding interface* of a class, i.e., the set of all selectors that are implemented in the class and to which super sends are made by methods implemented in descendants. For example, **concept 105** shows that {$>$, $\geq$, $\leq$} is an important part of the overriding interface of *Magnitude*, since each of these selectors are overridden in descendant classes *Float*, *Double*, *SmallInteger* and *LargeInteger* for optimisation purposes.

The concept pattern can also detect situations of *implementation inheritance* [11]. Typically, when implementation inheritance is used, a class overrides many methods defined in its parent (and uses super sends to invoke the parent behaviour). Finally, the concept pattern can provide guidelines for framework customisation. If we define a new subclass of a given class, it is likely that we have to override the methods specified in the overriding interface of the parent class. For example, if we would decide to create a new subclass of *LimitedPrecisionReal* or *Integer*, it is very likely that we need to override all the selectors in {$>$, $\geq$, $\leq$}.

Concept Pattern 4: Local Self Send Captured in Descendants

A set of selectors $m_1 \ldots m_p$ are called via a *self* send in the class A and the selectors are implemented in A and in some of its descendant classes $B_1 \ldots B_k$. Figure 6 shows this concept pattern in a general way. It occurs in 31 concepts of the *Magnitude* concept lattice.

For example, **concept 69** has elements {*(Number,* {*raisedTo:, sqrt, ln, truncated*}*)*} and properties {*concreteSelfCaptureInDescendant: Float, concreteSelfCaptureInDescendant: Double, concreteSelfCaptureLocally: Number*}.

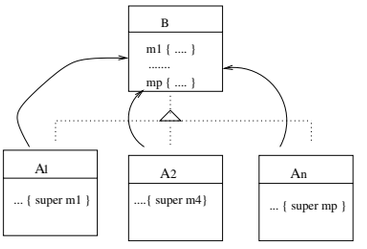


Fig. 5. Concept pattern 3

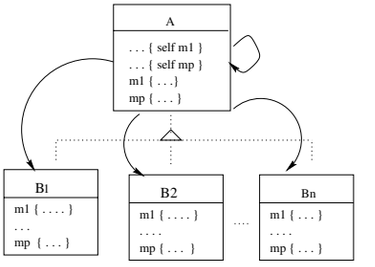


Fig. 6. Concept pattern 4

This concept pattern documents which specific methods are overridden in the subclasses of a common superclass. This means that the superclass defines some common or default behaviour for these methods, and each of the descendants can override this implementation via the mechanism of late binding with subclass-specific behaviour.

Concept Pattern 5: Local Self Send with Super Delegation

A set of selectors $m_1 \ldots m_p$ are called via a *self* and *super* send in a class A and the selectors are implemented in the same class A as well as in an ancestor class B. Figure 7 illustrates this concept pattern graphically. It occurs in 4 concepts of the *Magnitude* concept lattice. For example, **concept 48** has elements {*(SmallInteger,* {$>$, $\geq$, $\leq$}*)*} and properties {*(concreteSuperCapture: Magnitude, concreteSelfCaptureLocally: SmallInteger*}

This concept pattern documents delegation between methods in the same class and with the superclass. In all the found cases, the method that calls a selector via a *super send* has the same name as the selector itself. For example, in *SmallInteger* the method $\geq$ contains a "`super` $\geq$" statement. This means that part of the action to be executed (when a *self send* is made) is defined in the superclass, and the message is delegated by a *super send*.

Concept Pattern 6: Template Methods and Hook Methods

A set of selectors $m_1 \ldots m_p$ are called via a *self* send in a class A and the selectors are implemented as abstract methods in the same class A and are implemented as concrete methods in descendant classes $B_1 \ldots B_k$. Figure 8 illustrates this concept pattern graphically. It occurs in 7 concepts of the *Magnitude* concept lattice.

In the *Magnitude* hierarchy, this concept pattern only occurs for the subhierarchies with root classes *Integer* and *ArithmeticValue*. For example, **concept 31** has elements {*(ArithmeticValue,* {*, -}*)*} and properties {*abstractSelfCaptureLocally: ArithmeticValue, concreteSelfCaptureInDescendant:* {*LargeInteger, Fraction, Integer, SmallInteger, Float, FixedPoint, Point, Double*} }.

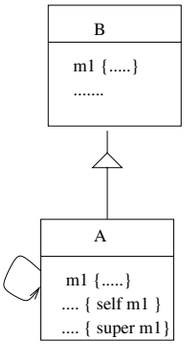


Fig. 7. C. pattern 5

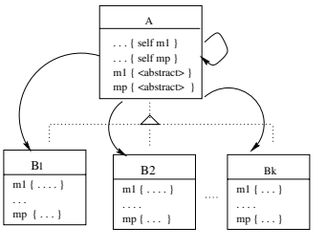


Fig. 8. Concept pattern 6

In this example, the abstract methods {*,-} in *ArithmeticValue* are called by other methods of the same class, but the actual implementation is defined in descendant classes. This concept pattern identifies the *hot spots* in an object-oriented application framework [9,3]. These hot spots are implemented by means of so-called *template methods* and *hook methods* [16,5]. In their simplest form, template methods are methods that perform self sends to abstract methods, which are the hook methods that are expected to be overridden in subclasses.

The information expressed in this concept pattern identifies the *abstract interface* of a class, as well as the subclasses that provide a concrete implementation of this interface. This information is essential during framework customisation when we want to add a *concrete* subclass of an *abstract* class, because it tells us which methods should be at least be implemented.

4 Related Work

Godin and Mili [7,8] used concept analysis to maintain, understand and detect inconsistencies in the Smalltalk *Collection* hierarchy. They showed how Cook's [2] earlier manual attempt to build a better interface hierarchy for this class hierarchy (based on interface conformance) could be automated. In C++ and Java, Snelting and Tip [13] analysed a class hierarchy by making the relationship between methods and variables explicit. They were able to detect design anomalies such as class members that are redundant or that can be moved into a derived class. The approach proved useful to serve as a basis for automated or interactive restructuring tools for class hierarchies. Siff and Reps [12] used concept analysis to modularise legacy C programs into C++ classes. Last but not least, Tonella and Antoniol [14] used concept analysis to infer structural design patterns from C++ code, which also provides crucial information to get a deeper understanding of object-oriented application frameworks.

All the above approaches only took information into account about which selectors are implemented by which classes. More behavioural information (e.g.,

based on self and super sends) was not considered. Hence, they could only detect *interface inheritance* but not *implementation inheritance*. As shown in this paper, more behavioural information about how a subclass is derived from its subclass is essential to analyse and understand the kind of reuse that is achieved.

5 Conclusion and Future Work

In this paper we analysed the well-known *Magnitude* inheritance hierarchy in Smalltalk using Concept Analysis. Based on information about self sends, super sends and invoked methods, we calculated the concept lattice for this hierarchy. We classified the generated concepts into *concept patterns*, which provide a roadmap of the code that ought to be analysed and understood. With the information given by the concept patterns, we discovered a number of interesting non-documented relationships about how classes and methods in the hierarchy are reused. A preliminary analysis of these patterns strengthened our belief that the technique is useful to: document the subclass interface of a class; provide guidelines on how an object-oriented framework can be customised or reused; identify hot spots in an object-oriented application framework; detect the type of inheritance (e.g. interface inheritance or implementation inheritance) used in an inheritance hierarchy; identify opportunities for refactoring; get insight in the potential impact of changes to framework classes. Based on these results, we believe that Concept Analysis is a promising technique in the understanding and re-engineering of large inheritance hierarchies.

Based on these results, we know that a lot of further research is necessary. One research avenue concerns the applicability of CA. We intend to confirm the usefulness of our method by analysing other well-known and non-trivial Smalltalk class hierarchies (e.g., Collection, Model, View and Controller). We also want to apply our approach to other object-oriented languages (such as Java and C++) to investigate the effect of language-specific properties (such as interfaces or multiple inheritance) by comparing similar class hierarchies in different languages. Another topic of future work is to investigate the effect of other behavioural information such as method invocations, variable accesses and variable updates; or the effect of other essential relationships between classes, such as composition and aggregation. Finally, we should take into account the additional information provided by how the concepts in the generated concept lattice are related via a partial order.

Acknowledgements

We thank Dirk Deridder, Oscar Nierstrasz, Roel Wuyts and the referees for their feedback.

References

1. G. Birkhoff. Lattice theory. *American Mathematical Society*, 1940. 54
2. William R. Cook. Interfaces and specifications for the smalltalk-80 collection classes. In *Proc. Int'l Conf. Object-Oriented Programming Systems, Languages, and Applications*, volume 27(10) of *ACM SIGPLAN Notices*, pages 1–15. ACM Press, October 1992. 61
3. Serge Demeyer. Analysis of overriden methods to infer hot spots. In Serge Demeyer and Jan Bosch, editors, *ECOOP '98 Workshop Reader*, volume 1543 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998. 61
4. Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999. 58
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems*. Addisson-Wesley, 1994. 61
6. B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999. 54
7. Robert Godin and Hafedh Mili. Building and maintaining analysis-level class hierarchies using galois lattices. In *Proc. Int'l Conf. Oject-Oriented Programs, Systems, Languages and Applications*, volume 28 of *ACM SIGPLAN Notices*, pages 394–410. ACM Press, October 1993. 61
8. Robert Godin, Hafedh Mili, Guy W. Mineau, Rokia Missaoui, Amina Arfi, and Thuy-Tien Chau. Design of class hierarchies based on concept (galois) lattices. *Theory and Application of Object Systems*, 4(2):117–134, 1998. 61
9. Ralph E. Johnson and Brian Foote. Designing reusable classes. *J. Object-Oriented Programming*, 1(2):22–35, February 1988. 61
10. John Lamping. Typing the specialization interface. In *Proc. Int'l Conf. Oject-Oriented Programs, Systems, Languages and Applications*, volume 28 of *ACM SIGPLAN Notices*, pages 201–214. ACM Press, October 1993. 57
11. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997. 59
12. Michael Siff and Thomas Reps. Identifying modules via concept analysis. In *Proc. Int. Conf. Software Maintenance*, pages 170–179. IEEE Computer Society Press, 1997. 61
13. Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. In *ACM Trans. Programming Languages and Systems*, 1998. 61
14. Paolo Tonella and Giulio Antoniol. Object oriented design pattern inference. In *Proc. Int'l Conf. Software Maintenance*, pages 230–238. IEEE Computer Society Press, 1999. 61
15. R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. *Ordered Sets, Ivan Rival Ed., NATO Advanced Study Institute*, pages 445–470, September 1981. 54
16. R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990. 61
17. R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proc. Int'l Conf. TOOLS USA'98*, pages 112–124. IEEE Computer Society Press, 1998. 56

Using Both Specialisation and Generalisation in a Programming Language: Why and How?

Pierre Crescenzo and Philippe Lahire

Laboratoire I3S (UNSA/CNRS)
Projet OCL 2000 route des lucioles
Les Algorithmes Bâtiment Euclide
B BP 121 F-06903 Sophia-Antipolis CEDEX, France
{Pierre.Crescenzo,Philippe.Lahire}@unice.fr
http://www.i3s.unice.fr/{~crescenz/,~lahire/}

Abstract. The reuse of libraries of classes by client applications is an interesting issue quite difficult to achieve, especially when modification of the class tree is needed but not possible because of the context. We propose a solution which is based on the presence of both specialisation and generalisation relationships in an object-oriented programming language. The specification of both relationships is based on a meta-model called *OFL* which provides a support for describing the operational semantics of a language through the definition of parameters and semantical actions. We propose an overview of the expressiveness of OFL and of its implementation and we give also some other interesting applications.

1 Introduction

In this paper we address the problem of the reuse of libraries of classes by client applications when modifications of the class tree is needed. We propose a solution which is based on the introduction of both specialisation and generalisation relationships in future object-oriented programming langages. This idea to combine both relationships altogether is also pointed out in [5] which focuses more on the integration feasibility within existing OOPL. According to the handling of libraries of classes there are other problems to solve like the maintenance of classes (removal of deprecated features, redefinitions, etc.) that may be solved using interclassing [4]. Even if those approaches deal with the use of libraries of classes by client applications, the philosophy is quite different: our approach deals with existing libraries that may not be modified by client applications whereas the other approach is related to the modification of libraries of classes themselves and their consequences in client applications.

To develop this idea, we present two main parts. Firstly, in section 2, we describe a very pragmatic situation where specialisation and generalisation are useful in the graph of types. You will see that the use of only one of them would lead to only a poor approximation.

Secondly, in section 3, we present a practical solution to define a new programming language with both specialisation and generalisation, or to improve an

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 64–73, 2002.

existing language with a reverse inheritance. This solution is based on the *OFL* Model ("Open Flexible Languages") [2]. The section 4 presents some implementation issues which handle principles of the *OFL* Model. Then, we conclude the paper in the last section, 5.

2 Why Both Specialisation and Generalisation?

Our approach is defined in the context where a programmer uses a software library of components (these components could be classes). He may have written this library or not, but he cannot modify it. This situation happens very often, for instance when the code is not provided, when it is *copyright*ed, when it has to be left unchanged for existing applications, and so on. The figure 1 give an example of such a very simple library with two very typical classes.

Now, for a specific program need or to make the library evolve, we want to add a component in the library (*i. e.* a class in the graph). This fact is illustrated in figure 2.

We can imagine three solutions to integrate Rectangle in the hierarchy:

1. The first is the most simple: "If we want to add a class, we must reorganise all the hierarchy!" This solution, illustrated in figure 3, is obviously the best one. But the best one if we can modify the hierarchy and an impossible one otherwise. And even if we could modify the existing classes, it could be a bad idea: we could add some bugs in some other applications which use these existing library and, in the example, the stability of Square is called into doubt by introducing Rectangle.

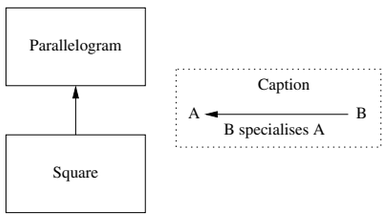


Fig. 1. An existing and unmodifiable hierarchy of classes

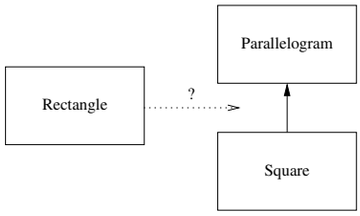


Fig. 2. A new class in the unmodifiable hierarchy

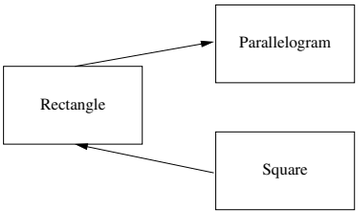


Fig. 3. The first solution: a total reorganisation of the hierarchy

2. A second solution respects the constraint of the unmodifiable existing hierarchy. The idea is to insert Rectangle as a specialisation of Parallelogram, as you can see in figure 4. Here there is no problem with existing classes and the relationship between Parallelogram and Rectangle is correct. But the instances of Square logically have to be instances of Rectangle and this is not the case here.

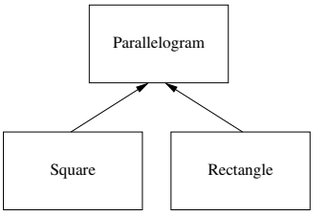


Fig. 4. The second solution: Rectangle specialises Parallelogram

3. The third solution is to take advantage of the fact that Rectangle is closer from Square than from Parallelogram. So, the idea is to specialise Square rather than Parallelogram as it is shown in figure 5. This solution is valid as long as polymorphism capabilities between Square and Rectangle are not used. The instances of Square logically have to be instances of Rectangle and this is the contrary here.

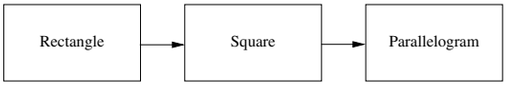


Fig. 5. The third solution: Rectangle specialises Square

As we just see, if we have only specialisation (the problem is the same if we have only generalisation), we can make evolution of a graph of classes without

risk (e.g. without modifying existing classes) but we can't have, simultaneously, a valid behaviour of the types (e.g. correct polymorphism capabilities) in the resulting graph.

Our proposition is to add a generalisation relationship in order to have both specialisation and generalisation in the same language. Generalisation is only the reverse link of specialisation so, theoretically, only one of them is sufficient. But practically, we could perfectly resolve our evolution problem with both. Let's show you a new figure, 6. It demonstrates a good way to handle evolution in our graph of classes. Rectangle is integrated as a specialisation of Parallelogram and a generalisation of Square. But what are the advantages in comparison with the three previous solutions?

The advantage of our solution in relation to the first one is that no class is modified in the initial graph. If we haven't the code or the right to modify it, we can nevertheless apply a relevant adaptation of the graph. And even if we can modify the initial graph, our solution protects the quality of Square since the new Rectangle must be compatible with the well-tried Square and not the contrary. In comparison with the second proposition, to use both links allows to make capital out of a correct behaviour of polymorphism between Square and Rectangle. In relation with the third approximative solution, the idea to use generalisation is better because the graph of types is relevant : a square is a rectangle and not the contrary! Obviously the pseudo-code inserted in figure 6 is not self-sufficient to explain the semantics and it should be deeply discussed. Particularly, he has to be linked to the possible parameter values presented in 3.2 and should be further specified by assertions built on the model reification. However, in order to give a flavour of the capabilities which may be provided to programmers, we could say that in a *Java*-like language, the two keywords *extends* and *generalizes* are strongly related to lookup operation (see 3.3) in order

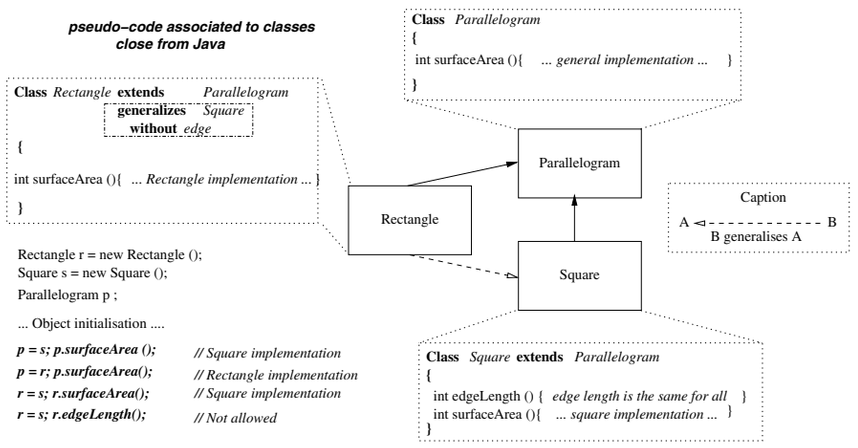


Fig. 6. A satisfactory integration of Rectangle with possible pseudo-code

to implement polymorphism and to allow to access to class instances as if the class hierarchy was the one described in figure 3.

3 How? The Model *Open Flexible Languages*

3.1 *OFL* in a Nutshell

This section presents the *OFL* model (Open Flexible Languages) and its capability to define easily both specialisation and generalisation.

The *OFL* Model aims to describe the main object-oriented programming languages (such as *Java*, *C++*, *Eiffel*, ...) to allow their evolution and their adaptation to specific programmer's needs. To reach this goal, *OFL* reifies all elements of an object-oriented programming language in a set of components of a language. Thus classes, methods, expressions, messages, and so on are the *OFL*-components and are integrated in a specific MOP (Meta-Object Protocol) which is self-extendable and contains the set of entities needed for the reification of both languages and user applications.

The meta-programmer creates a language by selecting adequate *OFL*-components in predefined libraries. He can also specialise a given *OFL*-component in order to generate one dedicated to some specific uses.

Classes are reified by *OFL*-components. Take the example of *Java*. We have ComponentJavaClass, ComponentJavaInterface, ComponentJavaArray, ... An originality of *OFL* is that relationships are also reified. So, we have for *Java*: ComponentJavaExtendsBetweenClasses, ComponentJavaImplements, etc[1].

To facilitate the creation of an *OFL*-component, *OFL* provides some meta-components, called *OFL*-concepts. So, we have a ConceptRelationship and a ConceptDescription (the word *description* has been chosen to represent classes and all entities which look like classes, such as interfaces). Thus, ConceptDescription as well as ConceptRelationship are equivalent to meta-meta-classes. In each concept, a set of parameters gives the meta-programmer necessary expressiveness to create or adapt an *OFL*-component.

3.2 Hyper-Generic Parameters

But how can the meta-programmer easily define the *OFL*-components for the language he wants to create or adapt? In fact, this work may be very difficult and tedious because he would have to rewrite a lot of algorithms such as type controls, dynamic links, use-of-polymorphism verifications, inheritance rules, etc..

In *OFL*, we provide a way to simplify this task: hyper-generic parameters. All the algorithms are predefined and are customized by hyper-generic parameters which have a value in each *OFL*-components.

In the sequel, we illustrate a subset of the hyper-generic parameters which can be applied to an *OFL*-component reifying a relationship to customize it. We explain each parameter and its capabilities of customization, and we give

[1] The full list of *OFL*-components for *Java* is given in [1].

its value when it is mandatory for the definition of ComponentSpecialisation and ComponentGeneralisation.

Kind In *OFL*, we handle four kinds of relationships: import for inheritance and all other importation links between descriptions, use for aggregation, composition, and all other use links between descriptions, type-object for all links between types and objects such as instantiation, and objects for all links between objects. Its value for ComponentSpecialisation and ComponentGeneralisation is import.

Cardinality It defines the maximal cardinality of a relationship. For example, the value of Cardinality is $1-1$ for a single inheritance and $1-\infty$ for a multiple one. We want to specify single links, so its value is $1-1$ for ComponentSpecialisation and ComponentGeneralisation.

Repetition It is useful if and only if Cardinality is not $1-1$ (to implement repeated inheritance, for example). For ComponentSpecialisation and ComponentGeneralisation, the value of Repetition is ignored.

Circularity It expresses if the *OFL*-component admits a circular graph (it is useful mainly for *use* relationships). Circularity is forbidden for ComponentSpecialisation and ComponentGeneralisation.

Symmetry This parameter points out if the *OFL*-component provides relationships that are symmetrical (e.g. a is-a-kind-of relationship). Neither ComponentSpecialisation nor ComponentGeneralisation is symmetrical.

Opposite We may have, in a language, two *OFL*-components with reversed semantics. This is an essential information for all actions which need to navigate through the graph of descriptions (e.g. to ensure type conformance). ComponentSpecialisation and ComponentGeneralisation are opposites.

Direct_access and Indirect_access These parameters give the capability to choose the policy of this visibility. In traditional inheritance, features of the ancestor are directly visible in the heir, as if they are declared in it. Some languages propose also to name the target-description for example to access to the old-version of a redefined method. For ComponentSpecialisation and ComponentGeneralisation it may be allowed or not[2].

Polymorphism_implication It can take four values: up means that all instances of the source-description must be also instances of the target-description; down points out the contrary (very useful to specify a generalisation); both means that source-description and target-description have the same instances (it is useful to describe versionning); none allows for example to define relationships dedicated to code reuse. The value for ComponentSpecialisation is up and it is down for ComponentGeneralisation.

Polymorphism_policy It indicates if a new declaration of attribute or method in the source-description hides the feature in target-description or overrides it. For ComponentSpecialisation and ComponentGeneralisation, we can use a traditional value: hiding for attributes and overriding for methods[3].

[2] All languages may not provide the same expressiveness. Just think about the difference about the handling of inheritance in *Java*, *C++*, *Eiffel* or *Sather*, etc.

[3] In *OFL*, overloading is not handled by relationships but by descriptions.

Feature_variance It proposes three kinds of variance rule for the redefinition of features : covariant like in *Eiffel* (the type indicated in the source-description must be the same or a subtype according to Polymorphism_implication), contravariant as in *Sather* [6] (this is the reverse), nonvariant as in *Java*[4] (the type indicated in the source-description must be the same than the one in the target-description). ComponentSpecialisation and ComponentGeneralisation do not impose a specific value but their status of opposite means a coordinated choice.

Assertion_variance It takes into account languages with assertions like *Eiffel*. It indicates the kind of variance for assertions: weakened (the assertion of source-description must be implicated by the assertion of target-description), strengthened (this is the reverse), unchanged (they must be equivalent).

Renaming, Adding, ... The First one is dealing with the right to rename a feature through a relationship defined by the *OFL*-component[5]. *OFL* also provides parameters to customize the capability to add, to remove, or to redefine assertions, method's signatures, method's bodies, and method's qualifiers, but also to mask, to show, to abstract, or to make effective the imported features. For example, according to ComponentSpecialisation, it is relevant to add a feature to a specialised class but not to remove any of them. To redefine a feature is also possible. ComponentGeneralisation should have the opposite semantics : for instance we should be able to remove but not add some feature.

3.3 Actions

To associate values to a set of parameters may be appropriate for describing the customized behaviour of a sort of relationship. But we need more to allow relevant control and execution of these links so that *OFL* includes a list of actions.

Each action defines the operational semantics of a part of work traditionally handled during the compilation or execution time. And each action takes into account the value of the hyper-generic parameters. So the behaviour of the defined language is adapted to the value of each parameter of each component. We have classified our actions in seven categories: **actions to search a feature** (e.g. lookup to find the relevant feature in the graph of descriptions according to a message), **actions to execute a feature** such as execute which allows to perform a routine call, **actions to make a control** like are_valid_parameters which controls the compatibility between effective and formal parameters, **actions to handle instances of descriptions** (e.g. create_instance or destroy_instance), **actions to handle extension of descriptions** or **Base operation** such as assign or copy.

[4] If type of parameters of methods are not exactly the same, in *Java* this is overloading and not overriding.

[5] Renaming is possible in *Eiffel* but not in *Java* or *C++*.

How to Write an Action? An action could be simple such as verify_circularity which controls that all relationships with the parameter Circularity not set don't make circular graph. The algorithm of this action is simple: to go all over the graph for this relationship and to verify that none of the descriptions is direct or indirect target of itself. The moment to execute verify_circularity is also easy to imagine: it could be launch once in a static tool like a compiler or a code checker.

But other actions are a lot intricate! For example, let's examine the action lookup. To find the relevant feature in accordance with a message, the task may be difficult and the algorithm complicated. We have to take the value of many parameters into account. The value of Polymorphism_implication is used to build a graph of types. Polymorphism_implication will help to determine which policy (hiding or overriding) have to be considered. With Cardinality and Circularity, we can choose an efficient way to go all over the graph. Symmetry could help us to adopt a two-direction route. Direct_access and Indirect_access give information about the visibility of the target-description. Finally, Feature_variance, Assertion_variance, Adding, Removing, and so on, allow to know how features are imported. Furthermore, the moment when it is correct to execute the lookup is also not obvious. We can easily imagine that a first part of this task is static (determination of all unambiguous calls for example) and another one is dynamic (dynamic linkage at runtime for example).

Then we may understand that it is possible to write the code of lookup. But if we want to provide some useful model to the programmer, it is obviously necessary to help him to write actions. In this way, we supply three things. The first one is that we have split complex actions in more elementary ones[6]. For example, we have a local_lookup which make the local (independently of all import relationships) research of a relevant feature in a description and a match which takes a feature and a message and determines if the second one is compatible with the first one... Thus, we split the difficulty of the complex lookup which has to call local_lookup, match and other actions to make its job.

Secondly, to solve the problem of the static and dynamic facets of our actions, we provide a way to define them in several parts. So, in fact, each action is split in a set of facets and each facets is declared static (used in a preliminary step like a compiler, a code checker, or a first access) or dynamic (used in an interpreter, an execution engine, or a virtual machine).

Thirdly, we intend to provide some default behaviours for all actions. Indeed, *OFL* could be used for a large variety of tools about source code. In this paper, we present a way to assist an extension of a programming language, but it is also possible to use our actions to make others tools like a code checker, some trace service, or a wizard for programming. So, our idea is to write typical algorithms for actions and to supply them in libraries.

Who Writes the Actions? There are three possible answers to this question. As we just explained, *OFL*-designers (we) have to provide libraries of actions

[6] Chapter 6 of [2] presents more than fifty actions.

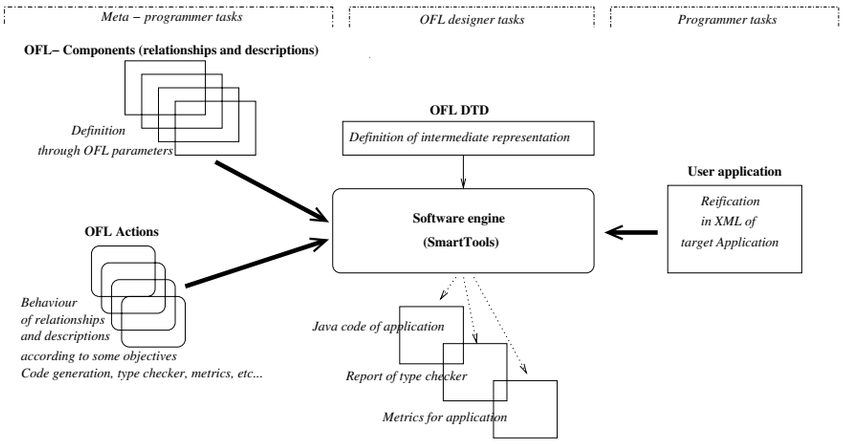


Fig. 7. Architecture of OFL implementation

for the more frequent usages. These libraries must be for very general purposes. When a relevant solution is not given in these libraries, the meta-programmer (the person who designs a language or a tool for handling source-code) has to redefine some of the actions or, in a bad case, to rewrite all of them. It is here useful to create a kind of plug-in library which adds some interest to the *OFL* set of tools. Finally, when the meta-programmer wants to add a very particular behaviour, he can redefine or write some actions in order to handle this behaviour. As this case is for a specific use (useful for an unique application, for example), creation of a library is not useful and the redefinition could be temporary.

4 Implementation Issues

Firstly, an implementation of *OFL* (cf. fig. 7) is based on the reification of both language semantics (*OFL*-components instances of an *OFL*-concepts) and application entities such as method, attribute, statement, etc. Because it is not reasonable to design a reification which deal with any entity of any language, it is necessary to design an extensible reification model. All this issues are achieved through a Meta-Object Protocol (MOP) written in *Java* and called *OFL/J*. In order to make easier the coupling with other tools, an XML-DTD of *OFL/J* can be generated automatically whereas meta-information and application reification are stored under an XML representation which conforms to this DTD.

Secondly, the reification of application should be parsed and semantics actions should be performed on each entities according to the language semantics. This will be done by *SmartTools* [3] which allows to define visitors (design-pattern) in order to allow the description of semantical actions to be associated

to application entities. *SmartTools* apply all these actions, automatically to any node of the abstract syntax tree associated to the application reification.

One interesting thing is the flexibility of the system. Actions can be added or removed from *OFL/J* and they can implement the approach described above from different point of view: to control the appropriateness between the body of application methods and the relationships defined between the classes within the reification, to generate pure *Java* code according to the information above, or to do both control and generation. Many other variants may be found according to the level of reification of statements and expressions (e.g. to insert into action-semantics the code for implementing an open virtual machine).

5 Conclusion and Future Work

In this paper we demonstrated that it was interesting to make coexist both specialisation and generalisation relationships, in order to better handle libraries of classes in the design of application. Other relationships also could be useful such as a reuse-code relationship whose aim is to provide one class the capability to include some methods from existing classes without allowing any polymorphism for its instances with those classes. These are only examples of the kind of relationships that *OFL/J*, the implementation of *OFL* model could handle. the part of *OFL/J* which deals with meta and non meta information reification and with the *OFL* Mop for extending the capabilities of the reification are implemented. Now we are investigating how to implement a first version of the semantics actions into *SmartTools*.

References

1. A. Capouillez, P. Crescenzo, and P. Lahire. Le modele OFL au service du meta-programmeur - Application à Java. In *LMO'2002*. Hermes Sc Pub., *L'objet*, vol. 8, N° 1-2/2002, Jan. 2002. 68
2. P. Crescenzo. *OFL : un modele pour parametrer la semantique operationnelle des langages a objets - Application aux relations inter-classes*. PhD. Thesis, University of Nice-Sophia Antipolis, December 2001. 65, 71
3. D. Parigot. Web Site of *SmartTools*. World Wild Web, Dec. 2001. `http://www-sop.inria.fr/oasis/SmartTools/`. 72
4. P. Rapicault and A. Napoli. Evolution d'une hierarchie de classes par interclassement. In *LMO'2001*. Hermes Sc. Pub., *L'objet*, vol. 7, N° 1-2/2001, jan. 2001. 64
5. M. Sakkinen. Exheritance - Class Generalisation Revived. In *ECOOP'2002 (The Inheritance Workshop)*, jun. 2002. 64
6. D. Stoutamire and S. Omohundro. Sather Specification. Technical report, International Computer Science Institute, University of Berkeley, Aug. 1996. 70

Automatic Generation of Hierarchical Taxonomies from Free Text Using Linguistic Algorithms

Juan Llorens[1] and Hernan Astudillo[2]

[1] Computer Science Department, Universidad Carlos III de Madrid
Leganes, Madrid, Spain
llorens@inf.uc3m.es
[2] Financial Systems Architects
25 Broad St., New York, NY, USA
hernan@acm.org

Abstract. This paper presents a technique, based on linguistic algorithms, to construct hierarchical taxonomies from free text. These hierarchies, as well as other relationships, are extracted from free text by identifying verbal structures with semantic meaning. The created taxonomies can be used as kernel for complete domain representation of a particular knowledge area. The domain representation could allow software architects to reuse Information. In order to test the algorithms, an example domain is presented to exemplify the use of the technique.

1 Automatic Generation of Information Structures in Sw. Engineering

Software engineering (SE) has quickly evolved towards effectiveness in the lasts decade. Software projects development demand quality policies, metrics and best practices. Among them, software reuse has always been considered as a key factor to enhance an organization's productivity and strength. During the last years, the concept of software has evolved towards a more complete definition.

According to Jacobson et. al in [1], an artifact is "a tangible piece of information that (1) is created, changed, and used by workers when performing activities, (2) represents an area of responsibility, and (3) is likely to be put under separate version control. An artifact can be a model, a model element, or a document." Therefore, considering an artifact as the atomic element is software development, an artifact's reuse should no longer be just software reuse but information reuse, considering information as a general container of structured and free text data.

Implicit in this artifact definition, one of Software Engineering's main goals since the 1980's has been finding a way to represent software artifacts using a standard model. Software engineers needed a common language, and a common representation schema for visualizing, specifying, constructing, documenting, and communicating all kinds of artifacts [2]. At the end of the 1990's, the UML (Unified Modeling Language) [3] had become the clear standard, and many organizations have begun to

J.-M. Bruel and Z. Bellahsene (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 74-83, 2002.

use its information representation model to represent software artifacts. Unfortunately, the UML metamodel [4] has not been designed to cope with textual artifacts, particularly regarding textual information storage, which implies that a complete textual domain representation model can not be created to support information reuse.

Textual artifacts (text documents) were the first ones to be represented in a computer, as computers changed their goals from calculation to information processing [5]. Therefore, the first information representation models were developed to cope with this kind of artifacts. Many approaches to text representation models have been made, using free text words as descriptors, or using terms from controlled vocabularies [6] coming from basic taxonomies (Dewey decimal system, presented by Melvil Dewey in 1873 [7], [8]), or advanced ones, like semantic networks [9], Thesauri [10], [11], Topic Maps [12] or facets [13]. These controlled vocabularies are part of standard cataloguing practice in libraries and information traders and are now being applied to more advanced digital resources using thematic keywords in metadata resource descriptors. For example, the Dublin Core [14] standard metadata set includes elements for Title, Creator, Date, Format, etc. in addition to the more complex notion of the Subject (or theme) of a resource. Guidelines recommend that, where possible, the Subject element be taken from a relevant controlled vocabulary. Links between concepts in the subject domain can be expressed by the semantic relationships in a thesaurus. According to [10], the three main thesaurus relationships are Equivalence (equivalent terms), Hierarchical (broader/narrower, whole/part and enumerative terms), and Associative (more loosely Related Terms) [15].

However, most of the domain representation attempts, as DARE [16], DRACO [17], the intelligent libraries of Simos [18], Feature Oriented Domain Analysis (FODA) [19], the Synthesis project [20], the intelligent design of Lubars [21], the Rapid project [22], KAPTUR tool [23], Gomaa's domain analysis method [24] or the Organization Domain Modeling [25], provide a kind of support for hierarchical taxonomies as the kernel to organize information.

In order to support information reuse by software engineers, a clear domain representation is needed; the lack of such is one of the essential drawbacks towards the generalization of reuse practices inside the software development process. In order to settle the issue, classification algorithms assisting automatic generation of domain representations have been tried in the last years, coming from such disciplines like pattern clustering (from artificial intelligence), data-text mining, psychology or information science.

Many researchers have been trying to apply cluster analysis and pattern matching algorithms (originally from Artificial Intelligence) to information classification. Neural networks like Kohonen's maps [26], ART based algorithms and classical perceptrons [27], have been used. The main application has been to find clusters of semantically common terms, linked by associative relationships.

Text mining has been another approach to information structuring. Although the main goal of miners used to be the location of associations within large data-text corpus, their work, specially in the application of complicated regression models, has led to interesting definitions for complex information structures. Relevant authors are Feldman & Hirsh [28] and Basili & Pazienza [29].

From a different perspective, studying the abstraction as a kernel principle, Rosch et. al. in [30] created their own models for information structure, including hierarchies.

Information science has provided plenty of algorithms for information classification. Main examples are Zipfás law [31] for information discrimination, Inverse Document Frequency (IDF) [32] for relevance measurements of terms or n-grams [33] for phrase filtering based on statistical calculations of grams (character groups) occurrences.

Also, co-wording algorithms [34] arise from some proposals related to bibliometrics. It consist basically in building up science maps by means of word occurrences. One common application of co-wording is Chen's algorithm [35]. This algorithm generates, for each pair of terms, a coefficient that measures the degree of relationship between them. The result of this algorithm is a set of terms, called cluster, that semantically are similar to each other. Again, the results of this algorithm are only associative relationships.

Statistics have also been widely used before, specially those dealing with classification. K-means (supervised), axial k-means used in [36] by Lelu, max-min or Isodata (not supervised) applied in [37], using Euclidean and Humming distance. It is clearly accepted today that statistical classifiers can not, without other information sources, get meaningful semantic clusters. Anyhow, these methods gather also associative relationships between terms.

Some authors have proposed methods to generate thesaurus structures, like Chen in [38], but the proposed techniques are only applied to the associative relationship of the ISO-2788 standard for information taxonomies (Thesauri). Diaz et. al proposed in 1998 [39] a recursive neural networks method to create hierarchies of clusters which would lead to a hierarchical taxonomy. Although it can be considered an advance, obtained results showed the need of an improvement in order to practically apply the algorithms.

As a complement to the experiments carried out in the last decade, Natural Language Processing (NLP) techniques started to be applied to information classification [40]. NLP provides accurate techniques to identify tags in natural language phrases. Modern translators have had high level of success in translating complex sentences.

The authors intention has been to apply NLP techniques to identify verb structures in order to find semantic information about the concepts the verb is linking.

2 Gen/Spe Hierarchical Relationships from Free Text

The main principle of the presented technique is to consider that hierarchical information regarding terms can, many times, be found explicitly in electronic documents, specially if the document corpus is carefully selected. Field description documents, manuals, state of the art summaries, etc. are a very good source to look for Generalization/Specialization (Gen/Spec) hierarchies. This technique identifies two types:

The main principle of the presented technique is to consider that hierarchical information regarding terms can, many times, be found explicitly in electronic

documents, specially if the document corpus is carefully selected. Field description documents, manuals, state of the art summaries, etc. are a very good source to look for Generalization/Specialization (Gen/Spec) hierarchies. This technique identifies two types:

2.1 Grammatical Gen/Spec Hierarchies Gathered from Term-Phrases

A normal term-phrase can certainly hide a specialization hierarchy, called grammatical in this work as it comes from a grammatical rule. For example, the term phrase "*Artificial Intelligence*" can provide the following taxonomical structure:

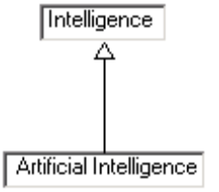


Fig.1. Taxonomical structure for the "Artificial Intelligence" phrase

2.2 Linguistic Gen/Spec Hierarchies

It is possible to identify Gen/Spec hierarchical relationships from text phrases by linking normalized verb structures to semantic relationships. For example, the following sentence: "*A MAZDA is a kind of car that [..]*" could provide the following hierarchy:

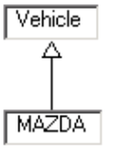


Fig.2. Specialization hierarchies gathered from linguistic parsing

The intention of the proposed technique is to identify verb structures in normal form (in the example, it would be to identify the "BE A KIND OF" verb expression) and to associate them to a semantic relationship (in this example, associate "BE A KIND OF" with a specialization hierarchy). This method allows not only to identify specialization hierarchies but also aggregations, compositions, associations etc. This feature is the essence of its main application: to generate representations of domains. The algorithm, which indeed is a kind of indexer, is presented in the following diagram:

This indexer is called RelationSHiPs Indexer (RSHPIndexer) because it needs to identify relationships. Finally, to construct a complete taxonomy from the { <Name>, <Verb structure>, <Term> } tuples, a management process to link together all the

different hierarchies identified in the indexing process is needed. A brief description of the RSHPIndexer follows:

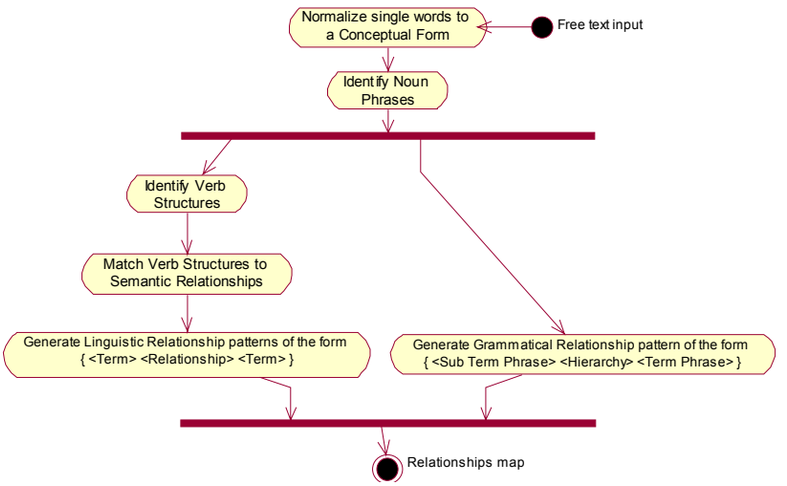


Fig. 3. Algorithm description using UML Activity diagrams.

A - Single Words Normalization into Conceptual Form: (*Worder*) The normalization process consists of checking whether a word is a generic form, and if it is not, transforming it into an accepted normal version of the representative concept, either in singular, plural, infinitive, etc. For example the input term □vehicles□ could be normalized to □Vehicle□ if this representation is considered normal for the <vehicle> concept. If the generic form of a given word input can not be found, that word is included as a new generic form of a new concept. This process is based on classical stemming algorithms. The single word normalization has been fully presented in [41].

B – Noun Phrase Identification: *(Phrase Normalizer)* The Phrase Normalizer intends to identify either noun and verbal phrases. Single terms can be used to represent concepts but usually, the more specific a domain is (or its taxonomic representation) the more noun phrases are used. In order to identify noun phrases, the following automaton was defined:

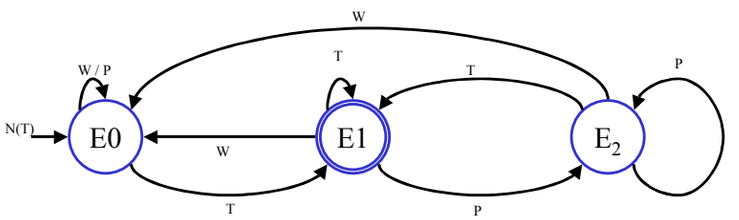


Fig. 4. Noun Phrase identification automaton

where: ***T:*** Noun Terms, ***P:*** Noun connectives, ***W***: Rest of grammatical tags, ***N(T)***: Automaton start

This automaton can be evolved to a more complicated one, according to the particular needs. The ***E1*** state is the only one returning results. It returns the concatenated text chain in the form of a Noun Phrase.

C - Verb Structures Identification: ***(Phrase Normalizer)*** Essentially, the previous automaton can be reconfigured for identifying verb structures, in the type of "BE A KIND OF"

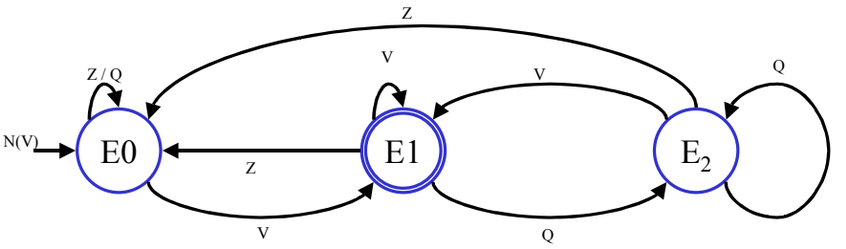


Fig. 5. Verb structures identification automaton

where: ***V:*** Verb Terms, ***Q:*** Verb connectives, ***Z***: Rest of grammatical tags, ***N(V)***: Automaton start The ***E1*** state is the only one returning results. It returns the concatenated text chain in the form of a Verb Structure.

D -Verb Structures Matching Against Semantic Meaning This stage is a simple mapping between language text chains and relationship meanings. Text chains represent verb structures, and relationship meanings represent semantic links between concepts. This step is quite critical as it intends to map syntactical analysis with semantic meaning. This information is stored in Database tables. For example:

Verb Structure	Relationship Type
<BE A KIND OF>	Generic- Specific hierarchy
<BE A TYPE OF>	Generic- Specific hierarchy
<BE A PART OF>	Whole Part Aggregation (Whole/Part hierarchy in ISO-2788 standard)
<BE EQUIVALENT TO>	Equivalence relationship (synonym in ISO-2788 standard)

etc.

E - Identification of Linguistic RelationSiHP Tuples: ***(RSHPExtractor)*** Once a verb structure has been matched against a relationship type, a possible Relationship can be created, depending on the noun phrases position around the verb structure. Therefore, in order to identify the whole relationship tuple, {<Noun phrase> , <Verb structure>, <Noun phrase>} from the Noun and Verb phrases, a third automaton was designed :

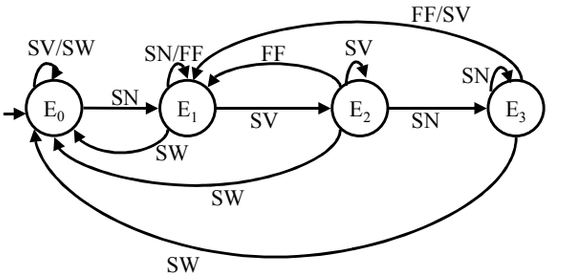


Fig. 6. Relationships generation automaton

where: ***SV:*** Verb Structure, ***SN:*** Noun Phrase, ***SW***: Stop Word, , ***FF:*** End of Phrase. The ***E3*** state is the only one returning results. It returns the relationship tuple.

F - Identification of Grammatical RelationSiHP Tuples: (RSHPExtractor) The algorithm has been designed to extract grammatical relationships from Noun Phrases.

3 Conclusions

This paper presents a technique which main goal is to identify relationships from free text using linguistic algorithms. Although the technique extracts all kinds of relationships, we have concentrated the presentation on the issue of hierarchical relationships. Two different types of sources were used to identify them: the noun phrases and the verbal structures, all of them found within free text electronic documents. An experiment was made to test the algorithm, showing that it extracted a low number of hierarchies. The main problems detected point to the need of a complete semantic ontology for verbal structures, which would help to solve the "gray-zone problems" formed by hierarchical Gen/Spec, composition and aggregation. However, the main goal of this algorithm was fulfilled, namely helping domain engineers to represent domain structures using taxonomies, since the computer provides the expert with taxonomic structures, albeit some of them are incomplete or incorrect, which can be used as start point for more complex domain generation procedures.

Acknowledgments

The authors would like to acknowledge the importance of Irene Diaz PhD study at the root of this work [42]. The algorithm presented here has evolved from experiments by Irene and us at the end of the 90's. Her (hopefully provisional) retirement from research activities in this area is sad news for the community.

The work presented in this paper was funded by the Technology and Science Ministry of Spain - MCYT, in the PROFIT program titled "Sistema de Gestin del Conocimiento en Calidad Desarrollado en Espaol y en Ingls."

References

1. Jacobson, I., Booch, G., Rumbaugh, J. (1999): The unified software development process. Reading (Massachusetts), Addison-Wesley.
2. Rumbaugh, J., Jacobson, I. & Booch, G. (1998): The unified modeling language reference manual. Addison-Wesley.
3. UML, 2001: http://www.uml.org/
4. OMG Unified Modeling Language, Specifications V 1.4. Semantics. http//www.omg.org/technology/documents/formal/uml.htm
5. Mooers, C.N. (1950): Information Retrieval viewed as temporal signaling Proceedings of the International Conference of Mathematicians, Cambridge, Massachusetts. 1950.
6. Luhn, H.P. (1957): A statistical approach to mechanized encoding and searching of literary information. IBM Journal of Research and Development, 1.
7. Dewey, M. (1979): Decimal Classification and Relative Index. Forest Press Inc.
8. On-Line Computer Library Center, Inc. <http://www.oclc.org/dewey/about/index.htm>
9. Quillian, M. R. (1968): Semantic Memory In: Semantic Information Processing, M. Minsky (ed.).
10. ISO-2788, 1986: Guidelines for the Establishment and Development of Monolingual Thesauri. International Organization for Standardization, 2nd edition -11-15 UDC 025.48. ISO 2788. Geneva.
11. ISO-5964, (1985): Guidelines for the establishment and development of multilingual thesauri. International Organization for Standardization, ISO 5964:1985., Geneva.
12. ISO/IEC 13250 (2000): Information technology - SGML Applications - Topic Maps, Geneva: ISO. February 2000.
13. Ranganathan, S. R. Prolegomena to Library Classification. Asian Publishing House. India. 1967.
14. http://dublincore.org/
15. "Semantically Indexed Hypermedia: Linking Information Disciplines", Douglas Tudhope and Daniel Cunliffe, 2000. ACM Computing Surveys. http://www.cs.brown.edu/memex/ACMCSHT/6/6.html.
16. Frakes, W & Prieto-Díaz, R, Fox, C.: DARE: Domain Analysis and Reuse Environment Annals of Software Engineering, (5) 125-141, The Netherlands: Baltzer Science Publishers, 1998.
17. Neighbors, J.: The Draco Approach to Constructing Software from Reusable Components IEEE Transactions on Software Engineering. Vol. SE-10 (5), 1984.
18. Simos, M.: The Growing of an Organon: a Hybrid Knowledge-based Technology and Methodology for Software Reuse In: Domain Analysis and Software Systems Modeling, (eds.) R. Prieto-Díaz & G. Arango, IEEE Computer Society Press, pp 204-221, 1991.
19. Kang, K., Cohen, S., Hess, J., Novak, W. & Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, 1990.

20. Campbell, G., Faulk, S. & Weiss, D.: □Introduction to Synthesis□ Technical Report Intro_Synthesis-90019-N, Software Productivity Consortium, Herndon, 1990.
21. Lubars, M.: □Domain Analysis and Domain Engineering in IdeA□ In: Domain Analysis and Software Systems Modeling, (eds.) R. Prieto-D□az & G. Arango, IEEE Computer Society Press, pp.163-178, 1991.
22. Vitaletti, W. & Guerrieri, E.: □Domain Analysis within the ISEC RAPID Center□ Proceedings of the Eighth Annual National Conference on Ada Technology, 1990.
23. Bailin, S.: Domain Analysis with KAPTUR. Course Notes. CTA Inc. Rockville, 1992.
24. Gomaa, H., Kerschberg, L., Sugumaran, V., Bosch, C. & Tavakoli, I.: □A Prototype Domain Modeling Environment for Reusable Software Architectures□ 3rd International Conference on Software Reuse, pp 74-83. IEEE Computer Society Press. Rio de Janeiro, 1994.
25. Simos, M.: □Organization Domain Modeling (ODM): Formalizing the Core Domain Modeling Life Cycle□ SIGSOFT Software Engineering Notes. Special Issue on the 1995 Symposium on Software Reusability, 1995.
26. Kohonen, T.: □Self-organized formation of topologically correct feature maps□ Biological Cybernetics, 43:59-69, 1982.
27. Rosenblatt, F.: □The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain□ Psychological Review, 65, 386-408. 1958.
28. Feldman, R. ; Hirsh , H. Mining Associations in Text in the Presence of Background Knowledge. Proc. of the 2nd Int. Conf. on Knowledge Discovery and Data Mining (KDD-96), 343-346. AAAI Press, 1996.
29. Basili, R. & Pazienza, M. T. Lexical Acquisition for Information Extraction. Information Extraction: A Multidisciplinary Approach to an Emerging Information Technology (M. T. Pazienza, ed.), (Frascati, Italy), LNAI Tutorial, Springer, July 1997.
30. Rosch, E., Mervis, C. B., Gray, W., Johnson, D., & Boyes-Braem, P. Basic objects in natural categories. Cognitive Psychology, 8, p382,439.-1976
31. Zipf, G. K.: Human Behaviour and the Principle of Least Effort: An Introduction to Human Ecology. Haffner. New York, 1972.
32. Spark Jones, Karen: □A statistical interpretation of term specificity and its application in retrieval.□Journal of Documentation, 28 (1972), 11-21.
33. Cohen, J. D.: □Highlights: Language and Domain Independent Automatic Indexing Terms for Abstracting□ Journal of the American Society for Information Science, 46(3), pp 162-174, 1995.
34. Gallant, S. (1995): Neural Network Learning and Expert Systems. The MIT Press.
35. Chen, H., Yim, T., Fye, D. & Schatz, B. (1995): □Automatic Thesaurus Generation for an Electronic Community System□ Journal of the American Society for Information Science. Vol. 46(3).
36. Lelu, C. (1993) Mod□les neuronaux pour l□analyse de donn□es documentaires et textuelles. Ph. D. Universit□de Paris.
37. Gallant, S. (1995): Neural Network Learning and Expert Systems. The MIT Press.

38. Chen, H., Yim, T., Fye, D. & Schatz, B. (1995): "Automatic Thesaurus Generation for an Electronic Community System" Journal of the American Society for Information Science. Vol. 46(3).
39. Díaz, I., Velasco, M., Llorens, J., Martínez, V.: "Semi-Automatic construction of thesaurus applying domain analysis techniques". International Forum on Information and Documentation. October 1998.
40. Mahesh, K. & Nirenburg, S.. Semantic classification for practical natural language processing. Proceedings of Sixth ASIS SIG/CR Classification Research Workshop: An Interdisciplinary Meeting, Chicago IL, October 8, 1995.
41. Diaz, I., Llorens, J., Morato, J.: "An Algorithm for Term Conflation Based on Tree Structures" Journal of the American Society for Information Science and Technology (JASIST), 53(3), 2002.
42. Díaz, I: "Esquemas de Representación de Información basados en Relaciones. Aplicación a la Generación Automática de Representaciones de Dominios". PhD Thesis. Universidad Carlos III de Madrid. Leganés, Madrid (Spain), 2001.

Guessing Hierarchies and Symbols for Word Meanings through Hyperonyms and Conceptual Vectors

Mathieu Lafourcade

LIRMM, Montpellier, France
{lafourca}@lirmm.fr

Abstract. The NLP team of LIRMM currently works on lexical disambiguation and thematic text analysis [*Lafourcade*, 2001]. We built a system, with automated learning capabilities, based on conceptual vectors for meaning representation. Vectors are supposed to encode *ideas* associated to words or expressions. In the framework of knowledge and lexical meaning representation, we devise some conceptual vectors based strategies to automatically construct hierarchical taxonomies and validate (or invalidate) hyperonymy (or superordinate) relations among terms. Conceptual vectors are used through the thematic distance for decision making and link quality assessment.

1 Introduction

In the framework of meaning representation, the NLP team of LIRMM currently works on strategies for automatically populating hierarchical taxonomies. Such strategies are based on the simultaneous exploitation of the conceptual vector model, definitions found in human usage dictionaries, and free text. The conceptual vector model aims at representing thematic activations for chunks of text, lexical entries, locutions, up to whole documents. Roughly speaking, vectors are supposed to encode *ideas* associated to words or expressions. The main applications of the model are thematic text analysis and lexical disambiguation [*Lafourcade*, 2001] and can found interesting approaches for vector refinement through the lexical implementation of taxonomies. Practically, we have built a system, with automated learning capabilities, based on conceptual vectors and exploiting monolingual dictionaries for iteratively building and refining them. So far, from French, the system learned 87000 lexical entries corresponding to roughly 350000 vectors (the average meaning number being 5). We are conducting the same experiment for English.

With these lexical and vector resources, we can, in conjunction with simple hyperonym (or superordinate) extraction methods [*Hearst*, 92], automatically construct many partial hierarchies. In our context, the hyperonymy relation is (perhaps abusively) considered as the inverse of the *hyponymy* relation, more often refered in software engineering as *specialization*. The *hierarchy soup* composed of hierarchy fragments is built through an iterated process that involves

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 84–93, 2002.

several strategies. The ideas, applied to French in our experiment, are generic and could be extended to any language. The bootstrapping consists in producing a set of hyperonyms from definition dictionaries that are corresponding directly to meanings as defined in our French dictionary. Filtering and selection are done with the help of thematic distance on the vectors associated to the items. The adjunction of hyponyms extracted from definitions, permits to add new meanings or salient properties to the hierarchies. At least, it allows us to strengthen our links or to detect inconsistencies. Free texts can also be exploited although (contrary to entry definitions), in case of polysemy, a word meaning identification should be carried out.

Beside NLP, taxonomy extraction can find applications in intelligent assistance in domain modeling and software engineering. This is specially critical, when (at least) two sets of classes have to be merged, as strategies based uniquely on class definitions fall short because of their lack of interpreting capabilities of naming entities. The name of a class or of an attribute has normally been chosen by designers for their evocating power, and is definitively (at least) a very strong clue for semantic induction and (at most) sometimes the only information available [*Rayside, 2001*]. So far, automated strategies rely only on symbol matching but never on the semantic association carried by the symbols. Similarly to metrics used in software engineering and class hierarchy factorization [*Dao*, 01], the thematic distance helps evaluating similarity. The main difference between Software Engineering and Lexical Semantics remains for the latter that meaning is a blurred halo in the semantic space and is susceptible of slippage (through metaphor and meronymy, notably).

In this paper, we first expose the conceptual vectors model and the notion of semantic distance and contextualization. Then, we expose the hierarchy building strategies that associate meanings to hyperonyms through sets of correspondences and conceptual distances.

2 Conceptual Vectors

We represent thematic aspects of textual segments (documents, paragraphs, syntagms, etc.) by conceptual vectors. Vectors have been used in information retrieval for long [*Salton et MacGill*, 1983] and for meaning representation by the LSI model [*Deerwester et al*, 90] from latent semantic analysis (LSA) studies in psycholinguistics. In computational linguistics, [*Chauché*, 90] proposes a formalism for the projection of the linguistic notion of semantic field in a vectorial space, from which our model originates. From a set of elementary notions, concepts, it is possible to build vectors (conceptual vectors) and to associate them to lexical items. The hypothesis that considers a set of concepts as a generator to language has been long described in [*Rodget*, 1852] (*thesaurus hypothesis*). Polysemous words combine the different vectors corresponding to the different meanings. This vector approach is based on well known mathematical properties, it is thus possible to undertake well founded formal manipulations attached to reasonable linguistic interpretations. Concepts are defined from a thesaurus

(in our prototype applied to French, we have chosen [*Larousse*, 1992] where 873 concepts are identified). To be consistent with the thesaurus hypothesis, we consider that this set constitutes a generator space for the words and their meanings. This space is probably not free (no proper vectorial base) and as such, any word would project its meaning on this space.

2.1 Thematic Projection Principle

Let $\mathcal{C}$ be a finite set of n concepts, a conceptual vector V is a linear combination of elements c_i of $\mathcal{C}$. For a meaning A, a vector $V(A)$ is the description (in extension) of activations of all concepts of $\mathcal{C}$. For example, the different meanings of ‹*quotation*› could be projected on the following concepts (the *CONCEPT*[intensity] are ordered by decreasing values): V(‹*quotation*›) = *STOCK EXCHANGE*[0.7], *LANGUAGE*[0.6], *CLASSIFICATION*[0.52], *SYSTEM*[0.33], *GROUPING*[0.32], *RANK*[0.31], *ORGANIZATION*[0.30], *ABSTRACT*[0.25], ...

In practice, the larger $\mathcal{C}$ is, the finer the meaning descriptions are. In return, computer manipulation is less easy. It is clear, that for dense vectors[1] the enumeration of the activated concepts is long and difficult to evaluate. We would generally prefer to select the thematically closest terms, i.e., the *neighborhood*. For instance, the closest terms ordered by increasing distance of ‹*quotation*› are: $\mathcal{V}$(‹*quotation*›) = ‹*management*›, ‹*stock*›, ‹*cash*›, ‹*coupon*›, ‹*investment*›, ‹*admission*›, ‹*index*›, ‹*abstract*›, ‹*stock-option*›, ‹*dilution*›, ...

2.2 Angular Distance

Let us define $Sim(A, B)$ as one of the *similarity* measures between two vectors A et B (eq. 1), often used in information retrieval [*Morin*, 1999]. Then, we define an *angular distance* D_A between two vectors A and B (eq. 2). We suppose here that vector components are positive or null, and "$\cdot$" refers to the scalar product.

$$Sim(A, B) = \cos(\widehat{A, B}) = \frac{A \cdot B}{\|A\| \times \|B\|} \tag{1}$$

$$D_A(A, B) = \arccos(Sim(A, B)) \tag{2}$$

Intuitively, this function constitutes an evaluation of the *thematic proximity* and is the measure of the angle between the two vectors. We would generally consider that, for a distance $D_A(A, B) \leq \frac{\pi}{4}$, (i.e. less than 45 degrees) A and B are thematically close and share many concepts. For $D_A(A, B) \geq \frac{\pi}{4}$, the thematic proximity between A and B would be considered as loose. Around $\frac{\pi}{2}$, they have no relation. D_A is a real distance function. It verifies the properties of reflexivity, symmetry and triangular inequality. We can have, for example, the following angles[2] (values are in degrees):

[1] Dense vectors are those which have very few null coordinates. In practice, by construction, all vectors are dense.

[2] Examples are extracted from: `http://www.lirmm.fr/~lafourca`

$D_A(\text{‹}profit\text{›}, \text{‹}profit\text{›})=0°$ $D_A(\text{‹}profit\text{›}, \text{‹}product\text{›})=32°$
$D_A(\text{‹}profit\text{›}, \text{‹}benefit\text{›})=10°$ $D_A(\text{‹}profit\text{›}, \text{‹}goods\text{›})=31°$
$D_A(\text{‹}profit\text{›}, \text{‹}finance\text{›})=19°$ $D_A(\text{‹}profit\text{›}, \text{‹}sadness\text{›})=65°$
$D_A(\text{‹}profit\text{›}, \text{‹}market\text{›})=28°$ $D_A(\text{‹}profit\text{›}, \text{‹}joy\text{›})=39°$

The first value has a straightforward interpretation, as ‹*profit*› cannot be closer to anything else than itself. The second and third are not very surprising since a ‹*benefit*› is quite synonymous of ‹*profit*›, in the ‹*finance*› field. The words ‹*market*›, ‹*product*› and ‹*goods*› are less related which explains a larger angle between them. The idea behind ‹*sadness*› is not much related to ‹*profit*›, contrary to its antonym ‹*joy*› which is thematically closer (either because of metaphorical meanings of ‹*profit*› or other semantic relations induced by the definitions). The thematic proximity is by no way an ontological distance but a measure of how strongly meanings may relate to each others.

The graphical representations of the vectors of ‹*exchange*› and ‹*profit*› shows that these terms are indeed quite polysemous. Two other terms (‹*cession*› and ‹*benefit*›) seems to be more focused on specific concepts. These vectors are the average of all possible meanings of their respective word in the general Thesaurus [*Larousse*, 1992]. It is possible to measure the level of *fuzziness* of a given vector as a clue of the number of semantic fields the word meaning is related to.

Because of the vagueness related either to polysemy or to lacks of precision (only 873 general concepts), we have to *plunge* our vectors into a specialized semantic space. However, we cannot cut loose from the general ones for two reasons. First, even non-specialized words may turn out to be pivotal in word sense disambiguation of specialized ones. Second, we cannot know beforehand whether a given occurrence of a word should be understood in its specialized acception or more a general one.

2.3 Vector Operators

Vector Sum. Let X and Y be two vectors, we define their *normed sum* V as:

$$V = X \oplus Y \quad | \quad v_i = (x_i + y_i)/\|V\| \tag{3}$$

This operator is idempotent (we have $X \oplus X = X$). The null vector $\mathbf{0}$ is by definition the neutral element of the vector sum. Thus we write down $\mathbf{0} \oplus \mathbf{0} = \mathbf{0}$. We derive by deduction (without demonstration) the *closeness properties*

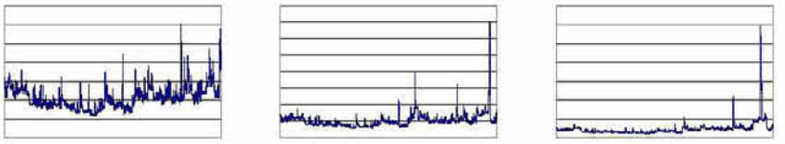


Fig. 1. Graphical representation of (more to less polysemous) terms *exchange*, *benefit* and *cession* (from left to right)

associated to this operator (both local and general closeness).

$$\begin{aligned} D_A(X \oplus X, Y \oplus X) = D_A(X, Y \oplus X) \leq D_A(X, Y) \\ \text{and} \qquad D_A(X \oplus Z, Y \oplus Z) \leq D_A(X, Y) \end{aligned} \tag{4}$$

Normed Term to Term Product. Let X and Y be two vectors, we define V as *their normed term to term product*:

$$V = X \otimes Y \quad | \quad v_i = \sqrt{x_i y_i} \tag{5}$$

This operator is idempotent ($X \otimes X = X$) and $\mathbf{0}$ is absorbent ($X \otimes \mathbf{0} = \mathbf{0}$).

Contextualisation. When two terms are in presence of each other, some of the meanings of each of them are thus selected by the presence of the other, acting as a context. This phenomenon is called *contextualisation*. It consists in emphasizing common features of every meaning. Let X and Y be two vectors, we define $\gamma(X, Y)$ as the contextualisation of X by Y as:

$$\gamma(X, Y) = X \oplus (X \otimes Y) \tag{6}$$

These functions are not symmetrical. The operator γ is idempotent ($\gamma(X, X) = X$) and the null vector is the neutral element ($\gamma(X, \mathbf{0}) = X \oplus \mathbf{0} = X$). We will notice, without demonstration, that we have the following properties of *closeness* and of *farness*):

$$D_A(\gamma(X, Y), \gamma(Y, X)) \leq \{D_A(X, \gamma(Y, X)), D_A(\gamma(X, Y), Y)\} \leq D_A(X, Y) \tag{7}$$

The function $\gamma(X, Y)$ brings the vector X closer to Y proportionally to their intersection. The contextualization is a low-cost meaning of amplifying properties that are salient in a given context. For a polysemous word vector, if the context vector is relevant, one of the possible meanings is *activated* through contextualization. For example, *bank* by itself is ambiguous and it vector is pointing somewhere between those of *river bank* and *money institution*. If the vector of *bank* is contextualized by *river*, then concepts related to finance would be considerably dimmed.

3 Hyperonym Identification and Hierarchy Construction

From term definitions found in human usage dictionaries and from free texts, we extract hyperonym and hyponym sets. The technic used is directly inspired from [*Hearst*, 92] for the use of simple and low cost pattern recognition. The main difficulty is that for a term t with k meanings, the proper word sense should be identified before figuring out its place in a hierarchy. When using (highly) specialized hierarchies [*Llorens*, 2001], we may skip this disambiguation process, although it still might lead to interpretation problems [*Barrière et al*, 01]. We use dictionary definitions and conceptual vectors for, at the same time (1) extracting the hyperonym from a well identified meaning, and (2) disambiguating the hyperonym candidate when needed.

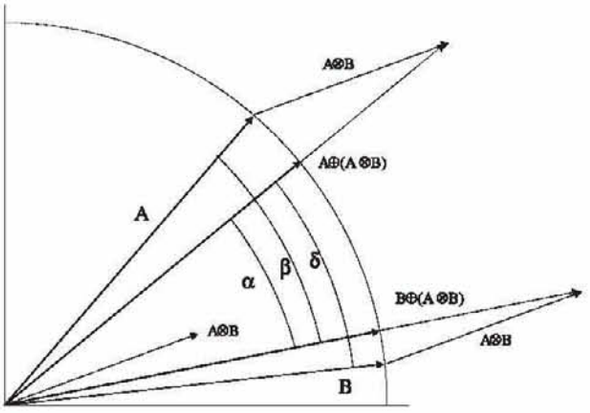


Fig. 2. Geometric represention (in 2D) of the contextualization function. The α angle represents the distance between A and B contextualized by each other

3.1 Hyperonym Identification

For a given term and from several vectorized dictionaries, we extract an hyperonym set. For instance, for the French term *émeraude* (emerald), we have two meanings with the following hyperonym sets:

- Hyper(émeraude.1) = pierre précieuse (src 1), pierre (src 1), [type de] béryl (src 2), gemme (src 3). (Eng. precious stone, stone, [kind of] beryl, gem.)
- Hyper(émeraude.2) = couleur [de l'émeraude], vert, vert lumineux (Eng. color [of emerald], green, shiny green)

The name of the source is given along the potential hyperonyms. Several candidates can be proposed for one definition as several patterns may be applied, and in general the frontier between under and over-contraction of definitions is dim. Parts of hyperonym between brackets are trimmed.

The difficulty here is to find one (or several) acceptable hyperonyms for each meaning. For each set, we compute the conceptual vector of each hyperonym candidate $V(\text{Hyper-cand/meaning}_i)$.

$$V(\text{Hyper-cand/meaning}_i)) = \gamma(V(\text{Hyper-cand}), V(\text{meaning}_i))$$

We contextualize the vector obtained from the definition of the hyperonym candidate with the vector of the definition it has been extracted from. If the hyperonym candidate exist as a term in the conceptual vector lexical database, then its vector is used. Otherwise, its vector is computed by composition of the vector of its sub-term (after a morphological and syntactical analysis).

The selected hyperonym candidate is the term which vector is the closest (in thematic distance terms) to $V(\text{Hyper-cand/meaning}_i))$. We are now able to create a node for each $\text{Hyper-cand/meaning}_i$ with a link to the corresponding disambiguated hyperonym candidate (cf Fig. 3 stage 1). If the term doesn't exist in the lexical database, it is added along its computed vector.

3.2 Identical Meaning Trimming

For a given meaning, we have added all disambiguated terms and links to hyperonym to the partial hierarchy. We need to delete all but one of the equivalent terms (cf Fig. 3 stage 2). The objective is to identify the most adequate item. Again, the strategy invoked here is straightforward, as only the link to the most specific (lowest in the hierarchy) is kept. In case of doubt (same level in the hierarchy or incompleteness of the hierarchy), the item which vector is the closest to its hyperonym is kept. In other words, only the most similar couple (term, hyperonym) is chosen.

This above procedure gives us a symbol for naming a given word meaning (this is useful only when the word is polysemous). This symbol is constructed with the simplest possible form: the concatenation of the term and of its most specific hyperonym. From a psycholinguistic point of view (which is out of scope here), the concision of the *symbol* would also be taken into consideration. Furthermore, such symbols are human readable and machine parsable. If the vector of *emerald/green* is not available, we (human and machine) can guess from the symbol that it might be a kind of green, and use the vector of *green* as a substitute. Of course, from the name of the hyperonym, which itself could be polysemous, we cannot without the hierarchy guess the proper meaning. But, in

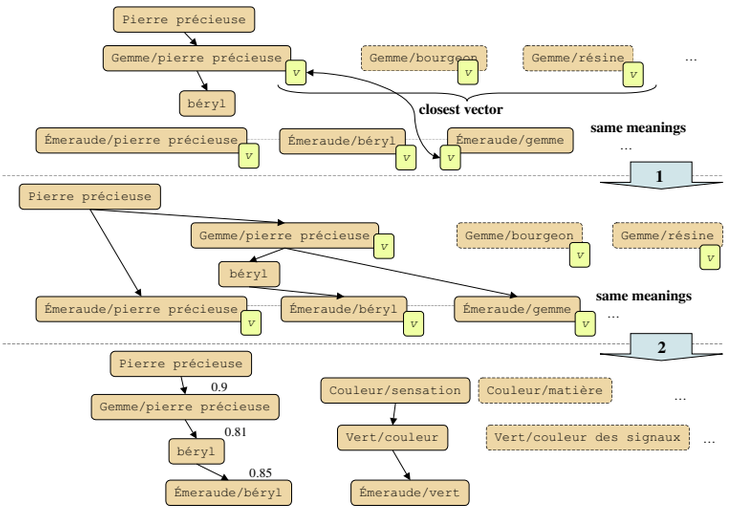


Fig. 3. (1) Linking of each meaning equivalent to its hyperonym. If an hyperonym is by itself ambiguous, its proper meaning is selected by minimizing the thematic distance between vectors. (2) Trimming of redundant meanings. When several meaning equivalents are competing, the one linked to the most specific hyperonym is selected. Other meanings are deleted as they related to upper items in the partial hierarchy

case the hierarchy is lost, the mutual information shared between the hyponym and the hyperonym would in most cases disambiguate both.

3.3 Hyponym Added Information

In dictionaries, many definitions of very general terms make extensive use of examples. Basically, these examples constitute hyponyms (the inverse relation relatively to hyperonyms) and could be exploited with benefit. The most obvious use of hyponyms is to cross-check hyperonyms, nevertheless we can also extract information that are not directly accessible from normal (hyperonymic) definitions. By the use of hyponymy, the hierarchy cannot take the form of a tree but of a (partial) lattice (cf Fig. 4). Indeed, a meaning can have several hyperonyms. For instance, we have extracted the following hyponyms (among others):

- Hypo(moyen de transport) = véhicule, voiture, avion, train, automobile, **cheval**, $\cdots$ (Eng. Hypo(transport means): vehicle, car, plane, train, motorcar, horse, $\cdots$)
- Hypo(viande) = poulet, agneau, boeuf, **cheval**, mouton, $\cdots$ (Eng. Hypo(meat): chicken, lamb, beef, horse, mutton, $\cdots$)

Here, we can observe that *horse* (cheval) is a particular *meat* (viande) and also a *means of transportation* (moyen de transport). Although, we have the following hyperonyms (familiar meanings excluded):

cheval.1: mammifère. (Eng. mammal)
cheval.2: art de monter à cheval. (Eng. art of ridding horses)
cheval.3: unitÄ de mesure. (Eng. measure unit)

We have seen clearly (through vector contextualization and thematic distance) that the two hyponym sets seem to induce two new meanings that were not given through definitions. In this case, we do create the new meanings (*cheval/moyen de transport* and *cheval/viande*) and link them to their hyperonyms. The problem is that starting from vectorized definitions, there is no way to catch these new meanings as they are not (yet) identified. Thus, to overcome this problem, we link each of these new meanings as hyperonym to its closest already existing counterpart. In the above example, we have:

- *cheval/moyen de transport* is closer to *cheval/mammifère* than to *cheval/unité de puissance*. This relation can be checked on their respective vector, and (sometimes) by pattern matching on some part of (encyclopedic) definition.
- *cheval/viande* is closer to *cheval/mammifère* than to *cheval/unité de puissance*.

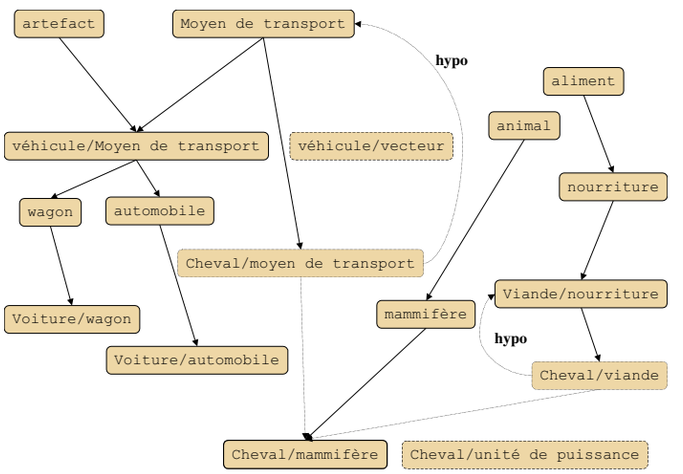


Fig. 4. Hyponym insertion. Adding found hyponyms can lead to the identification either (1) of new salient properties in already existing meanings or (2) of new meanings altogether. Thematic distance is used as a meaning selector

4 Conclusion

This paper has presented a strategy for hierarchy construction through low cost hyperonym extraction (from definitions) associated to disambiguation and linking decision based on conceptual vectors. By itself, the overall process consists in symbolizing word meaning (giving a unique name to each item that are members of a meaning set). It is now possible to handle a meaning not only by exemplifying its vector, but also by referring to it thanks to its symbol.

Our strategies have been prototyped and have been included in our (conceptual) vector lexical database. It is mainly used for comforting vector calculation and detecting inconsistencies. The overall process is by itself iterative and incremental. And a global hierarchy is being built by fusion of partial ones. Only some hierarchy parts are actually exploited during NLP process, mainly those which are really useful for word sense disambiguation. The experiment has been conducted (and is still in process) on 50000 nouns (for roughly 87000 words). Comforting enough is the fact the constructed hierarchy is really close to some Aristotelian classification. This is basically explained by the fact that the structure of the dictionary definitions draws much on this tradition. The main departure is the multiple inheritance schema that originates from property salience (as show on the term *horse*). In general, the frequency of this phenomenon is inversely proportional to the technicality of the domain.

The produced partial specialization hierarchies enable some automatic refinement of domain representation. In effect, when domains are too specialized, the fine meaning difference cannot be apprehended through conceptual vectors (unless enlarging considerably the vector space). Allowing agents to process in-

formations both at the vectorial and symbolic (as defined above) levels seems definitively a way to solve some aspects of the symbol grounding problem. Beside Natural Language Processing and information retrieval, possible application of this research is to provide intelligent assistance in advanced software engineering. Such assistance would mainly rely on guessing designer intentions through the inspection of names of entities. Relating lexical information to common knowledge could pave the way to more flexible domain representations.

References

[*Chauché*, 90] Jacques Chauché, *Détermination sémantique en analyse structurelle : une expérience basée sur une définition de distance.* TAL Information, 31/1, pp 17-24, 1990. 85

[*Barrière et al*, 01] Barrière C. and T. Copeck *Building Domain Knowledge from Specialized Texts* In Proc. of TIA 2001, 2001, 8 p. 88

[*Deerwester et al*, 90] Deerwester S. et S. Dumais, T. Landauer, G. Furnas, R. Harshman, *Indexing by latent semantic anlysis.* In Journal of the American Society of Information science, 1990, 416(6), pp 391-407. 85

[*Hamon*, 01] Hamon T. et A. Nazarenko *La structuration de terminologie: une nécessaire coopération* In Proc. of TIA 2001, 2001, 8 p.

[*Hearst*, 92] Hearst Marti A. *Automatic Acquisition of Hyponyms from Large Text Corpora.* In Proc. of the Fourteenth International Conference on Computational Linguistic COLING'92, 1992, Nantes, France, 8 p. 84, 88

[*Dao*, 01] Dao, M. M. Huchard, H. Leblanc, T. Libourel, C. Roume. *A new Approach to Factorization - Introducing Metrics* In Proc. of the METRICS 2002, 12 p. 85

[*Lafourcade et Prince*, 2001] Lafourcade M. et V. Prince *Synonymy and conceptual vectors.* Proc. of NLPRS'2001, Tokyo, Japan, August 2001, pp 127-134.

[*Lafourcade*, 2001] Lafourcade M . *Lexical sorting and lexical transfer by conceptual vectors.* Proc. of the First International Workshop on MultiMedia Annotation (Tokyo, Janvier 2001), 6 p. 84

[*Larousse*, 1992] Larousse. *Thésaurus Larousse - des idées aux mots, des mots aux idées.* Larousse, ISBN 2-03-320-148-1, 1992. 86, 87

[*Llorens*, 2001] Llorens J. and H. Astudillo *Automatic Generation of Hierarchical Taxonomies from Free Texts Using Linguistic Algorithms.* In Procs of MASPEGHI 2002, Lecture Notes in Computer Science, 7 p. 88

[*Morin*, 1999] Morin, E. *Extraction de liens sémantiques entre termes à partir de corpus techniques.* Thèse de doctorat de l'Université de Nantes, 1999. 86

[*Rayside, 2001*] Rayside D. and G. T. Campbell *An Aristotelian Understanding of Object-Oriented Programming* Minneapolis, Minnesota, October 2000. Edited by Doug Lea. pp 337- 353. 85

[*Rodget*, 1852] Rodget P. *Thesaurus of English Words and Phrases.* Longman, London, 1852. 85

[*Riloff*, 1995] Riloff E. and J. Shepherd *A corpus-based bootstrapping algorithm for Semi-Automated semantic lexicon construction.* In. Natural Language Engineering 5/2, 1995, pp. 147-156.

[*Resnik*, 1995] Resnik P. *Using Information contents to evaluate semantic similarity in a taxonomy.* In. Proc. of IJCAI-95, 1995, 8 p.

[*Salton et MacGill*, 1983] Salton G. et M. J. MacGill *Introduction to modern Information Retrieval* McGraw-Hill, New-York, 1983. 85

Reuse in Object-Oriented Information Systems Design

Daniel Bardou[1], Agnès Conte[1], and Liz Kendall[2]

[1] LSR-IMAG, France
[2] Sun Microsystems & Monash University, Australia

Preface

Improving reuse is still an important issue of Information Systems Engineering. Several current object-oriented approaches, such as patterns, frameworks or business components, address this topic at different phases of the software development process. This workshop focused on tools, techniques and methods developed to improve the reuse of design elements.

Reuse is today mainly gained with empirical tools and methods, and it is necessary to make reuse more systematic in object-oriented information systems design. Reuse at the design phase can be considered from two different yet complementary perspectives: (1) design for reuse or (2) design by reuse.

1. Design for reuse (1) deals with identifying reusable elements, specifying and organizing components, and integrating sets of
   and models for specifying reusable artifacts.
2. Design by reuse (2) needs to define new information systems engineering processes, and develop tools supporting systematic reuse of components in information systems.

Special attention has been given to contributions aiming to improve the above techniques by adapting novel or "non traditional" approaches at the edge of object-orientation (e.g. aspect-orientation or multi-viewpoints approaches, tools integrating artificial intelligence techniques, information retrieval, ...).

Many of the 6 papers selected for publication in these proceedings deal with the need to relate design activities with previous phases in the software development process. Adopting use-case patterns, a description-driven, a specification-oriented or a pattern language based approach are several possible ways to achieve this. Reuse at the design phase can of course also be eased by use of architectures and implementation techniques, such as XML-based middlewares that allows the collaboration of heterogeneous components, or HyperObjects that enable the handling of larger parts than single objects in an information system.

Having good reusable design artifacts is indeed not sufficient to achieve successful software reuse. Indexing and information retrieval techniques may help in searching the components that are the most suited to be used when designing a

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 94–95, 2002.

new information system. Reusable artifacts need to be adapted in most cases, and adaption techniques are an important issue, whatever the chosen technique (reflexive programming, framework instantiation, ...). New adaptation techniques and approaches, such as aspect-orientation, may for instance increase traceability of design components into programs. Further work needs to be achieved on these topics.

Organization

This workshop is organized by Daniel Bardou (LSR-IMAG, France), Agnès Conte (LSR-IMAG, France) and Liz Kendall (Monash University, Australia).

Program Committee

D. Bardou (LSR-IMAG, Fr)
C. Cauvet (Univ. Aix-Marseille, Fr)
S. Clarke (Trinity College, Ie)
A. Conte (LSR-IMAG, Fr)
J. Dospisil (Monash University, Au)
J. Han (Monash University, Au)
R. Johnson (Univ. of Illinois, USA)
L. Kendall (Monash University, Au)
M. Léonard (Univ. of Geneva, Ch)
A. Le Parc-Lacayrelle (LIUPPA, Fr)
W. Pree (Univ. of Salzburg, At)

Additional Referees

S. Turki (Univ. of Geneva, Ch)

Primary Contact

For more details on the workshop please contact:

Daniel Bardou
Equipe Sigma, LSR-IMAG & Université Pierre Mendès France
LSR-IMAG, 681, rue de la Passerelle, BP. 72
38402 Saint Martin d'Hères Cedex, France
email: Daniel.Bardou@imag.fr
tel: +33 (0)4 76 82 72 57
fax: +33 (0)4 76 82 72 87

Workshop website:
http://www-lsr.imag.fr/OOIS_Reuse_Workshop/

Software Reuse with Use Case Patterns*

Maria Clara Silveira[1] and Raul Moreira Vidal[2]

[1] Higher School of Business and Technology, Polytechnic Institute of Guarda
6300-559 Guarda, Portugal
silveira@fe.up.pt
[2] Faculty of Engineering, University of Porto
4200-465 Porto, Portugal
rmvidal@fe.up.pt

Abstract. This work concentrates on reuse-oriented software development. We propose an approach in which we incorporate reuse components in the initial phases of the software development process, that is to say, requirements specifications. These components, use case in the pattern form, reused during the requirements capture, allow a visualization of the system to be implemented. On the other hand, they also facilitate some normalization in the requirements process and, further, can be reused in several applications; this is obtained more easily with patterns. The primary motivation of this study derives from the fact that system requirements represent abstract knowledge of which a great part can be reused in other systems.

1 Introduction

Reuse development is now seen as a way to improve the productivity and quality of software systems, reducing both costs and risks. In other engineering areas, reuse of components is implicit, and even essential, in their training. The consistent and foreseeable stages of development reflect many years of experimentation and scientific development, contrary to what goes on in Software Engineering. We do not make trade-offs; we instead build systems to meet precise requirements thus making reuse a difficult practice [20].

A significant change in the way that software is conceived is, thus, fundamental. Reuse of software components embodies this change, completing the software development life cycle with new activities of reuse, including component classification, selection, composition and integration. Thus, the component-based software development life cycle differs from traditional software development in many ways [11,21]. The component-based software system design phase, for example, includes new activities such as selection and creation of software architectures, as well as selection

* Work developed with the support of the programme Prodep III (European Community, European Social Fund, Concourse 4/5.3/Prodep/2000, Request ref. 182.004/00-ESTG-IPG).

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 96-100, 2002.

and customisation of a set of software components [11]. This requires developing wrappers that glue reusable components together to build a software system [11].

Also, according to Meyer [18], the convergence of component-based development and quality is essential. On the other hand, the need for developing high-quality components to appreciate, criticize, and emulate is emergent.

Furthermore, reuse ought to be an ever-present concern in the life of a cost-effective organization, even if there is not an explicit reuse program [12]. However, reuse requires a substantial investment in an organizational infrastructure to encourage, support, maintain and manage the reusable components [24].

Although component-based development offers many potential benefits, such as greater reuse and a commodity-oriented perspective of software, it also raises several issues that developers need to consider. Developers must describe components in a way that is suitable for use by automated tools and is also understandable to human integrators [5]. Hence, developers may need to acquire new skills in document production, integration and maintenance [5].

In this context, we would like to propose use case components for reuse-oriented development. Use cases are pointed out as reusable components by several authors, like Biddle [4], D'Souza [8], Jacobson [14]. The use case approach provides a more natural partitioning of a complex system into more manageable pieces at the beginning of the development of the software system. Use cases can be seen as a sort of prototype on paper, describing, one by one, the various steps required to implement the system functionalities.

Use cases are a scenario-based technique for requirements elicitation which were first introduced in the Objectory method [13]. They have now become a fundamental feature of the UML notation for describing object-oriented system models.

The main purpose of the use case model is to define what the system should do, and to allow software engineers and customers to agree on this. The model consists of actor types, use cases, and relations between them [14].

The main motivation of this work derives from the fact that the system requirements represent abstract knowledge of which a great part can be reused in other systems. Comparative analysis of software systems has shown that, in general, 60 to 70 percent of systems functionality also exists in other systems. This includes code, design, functional and architectural similarities [16]. On the other hand, it is clear that much more than code should be reused; components, frameworks, documentation templates, software processes and development standards can all be reused [3].

2 Reuse-Oriented Development with Use Case Patterns

The architect Cristopher Alexander introduced the concept of pattern. He also [1] coined the term *pattern language* and explained the form well in his book [2]. A pattern language has the structure of a network. The elements of this language are entities called patterns [1].

Each pattern describes a problem that occurs over and over again in our environment and, then, describes the core of the solution to that problem [1]. Similarly, a software pattern describes a repeating problem in a specific context and a generic

solution for solving problem [6]. In other words, a pattern consists of four things: a title, a problem, a context, and a solution [7,10,15,23]. Additional information can be included, such as consequences, comments, and examples.

A common problem related to increasing the supply of well-documented patterns is the time required to capture them. *Pareto's Law*, also know as the 80/20 rule, states that 80 percent of the value is attributable to 20 percent of the effort [22, 23]. Coplien [7] has show us that it is possible to capture very important knowledge with significantly less effort by using a more compact pattern format. Thus it is important to pay attention to not spending extra effort in designing and constructing additional functionality, beyond the application's needs [22].

Software development is a highly repetitive activity, involving frequent use of patterns [17]. To go beyond their simple pedagogical value, patterns must go further. A successful pattern cannot just be a book description: it must be a software component, or a set of components [17]. These components can be adapted to various specific situations. In this sense, we would like to propose use case components in the pattern form, which will be referred to as use case patterns. These patterns can be documented much the same way as suggested for design patterns in Gamma [10]. Use case pattern is an approach to systematic use of existing patterns to ensure consistency of terminology.

In our approach, use case components can be reused during requirements capture by treating the components system as a toolbox of reusable elements [14]. Thus, use case components help the reusers as an entry point into the subsequent object models and implementation. These components can be documented. The terminology used to describe the components is very important. While it is important to document all reusable components, this is particularly crucial for actor and use case components, because these will be used in discussion with customers and other stakeholders, not just software engineers [14].

Therefore, all of this information requires the development of adequate databases of components, which interested developers may search by appropriate keywords to find out quickly whether some existing component satisfies a particular need. Having said that, we need a system that supports not only search and retrieval of reusable components but also librarian functions such as cataloging and classification.

A classic problem in software reuse is organizing collections of reusable components for effective search and retrieval. A software reuse library, organized around a faceted classification scheme, presents a partial solution to this problem [19]. This faceted scheme includes several dimensions still being studied. Faceted classification offers certain features that not only improve search and retrieval, but also support the potential reuser's selection process and contribute to the development of a standard vocabulary for software attributes [19].

The research and selection of the components from the repository, with the help of a tool, will be automatic, similar to the Patricia system [9]. This repository needs to be built and constitutes the primary difficulty in this approach.

The composition of the components is executed according to the needs of the customer; the customer's needs will also serve to verify the completeness of the solution. The customer validates and makes a decision depending on the availability of components.

In simple form, our proposed component-based development consists of six sequential major activities: identification and classification of components; research into the potential components for a project; composition and integration of the selected components; verification of the completeness of a given solution; creation and classification of new components; and optimization of the activities involved in the composition and integration of components.

3 Final Considerations

This work, although in a preliminary state of development, is a promising approach to component reuse, especially use case patterns. As use case pattern is an approach to systematic use of existing patterns to ensure consistency of terminology, the result is enhanced communication with stakeholders. By reusing the same use case components in various application systems, engineers will design systems in a uniform way.

After surpassing the problem of the use case patterns repository, we will test its applicability. Based on our experiences, we feel that the reuse of use case patterns in the initial phase of software development makes them ideal for capturing user's requirements, and hence systems constructing.

References

1. Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S.: A Pattern Language. New York: Oxford University Press (1977)
2. Alexander, C.: The Timeless Way of Building. New York: Oxford University Press (1979)
3. Ambler, S.: Reuse for the Real World. In Sdmagazine Online, April (2002)
4. Biddle, R., Noble, J, Tempero, E.: Supporting Reusable Use Cases. Victoria University of Wellington, New Zealand, in http://www.mcs.vuw.ac.nz/research, Technical Report CS-TR-01/10, October (2001)
5. Brereton, P., Budgen, D.: Component-Based Systems: A Classification of Issues. In Computer, Vol. 33, N°11, November (2000) 54-62
6. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern Oriented Software Architecture - a System of Patterns. John Wiley and Sons (1996)
7. Coplien, J., Schmidt, D. (Eds.): Pattern Languages of Program Design. Addison-Wesley Publishing Company (1995)
8. D'Souza, D., Cameron, A.: Objects, Components and Frameworks with UML. The Catalysis Approach. Addison-Wesley (1998)
9. Etzkorn, L., Davis, C.: Automatically Identifying Reusable OO Legacy Code. In Computer, Vol. 30, N°10, (1997) 66-71
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object Oriented Software. Addison-Wesley (1995)
11. Griss, M., Pour, G.: Accelerating Development With Agent Components. In Computer, Vol. 34, N°5, May (2001) 37-43

12. Herzum, P., Sims, O.: Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise. John Wiley & Sons (2000)
13. Jacobson, I., Christerson, M., Jonsson, P., vergaard, G.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley Publishing Company, Revised fourth printing (1993)
14. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process, Organization for Business Success. ACM Press, Addison-Wesley (1997)
15. Martin, R., Riehle, D., Buschmann, F. (Eds.): Pattern Languages of Program Design 3. Addison-Wesley Publishing Company (1998)
16. McClure, C.: Value-Added Year 2000: Harvesting Components for Reuse. Extended Intelligence, Inc., In http://www.reusability.com/papers2.html (1997)
17. Meyer, B.: Object-Oriented Software Construction. Second Edition, Prentice-Hall, Inc. (1997)
18. Meyer, B.: Software Engineering in the Academy. In Computer, Vol. 34, No. 5, May (2001) 28-35
19. Prieto-Dáz, R.: Implementing faceted classification for software reuse. In Communications of the ACM, Vol. 34, N 5, May (1991) 89 97
20. Prieto-Diaz, R.: Reuse in Engineering vs. Reuse in Software: Why Are They Incom-patible? Symposium on Software Reusability SSR 01, Tutorial/Conference Program, in http://www.abitmore.be/ssr2001/index.htm (2001)
21. Sommerville, I.: Software Engineering. Sixth Edition, Addison-Wesley (2001)
22. Sparling, M.: Lessons learned through six years of component-based development. In Communications of the ACM, Vol. 43, N 10, Oct (2000) 47-53
23. Vlissides, J., Coplien, J., Kerth, N. (Eds.): Pattern Languages of Program Design 2. Addison-Wesley Publishing Company (1996)
24. Yourdon, E., Whitehead, K., Thomann, J., Oppel, K., Nevermann, P.: Mainstream Objects: An Analysis and Design Approach for Business. Software AG, Prentice-Hall (1995)

Promoting Reuse through the Capture of System Description

Florida Estrella[1], Sebastien Gaspard[1,2], Zsolt Kovacs[3]
Jean-Marie Le Goff [3] and Richard McClatchey[1]

[1]CCCS, University of the West of England
Frenchay, Bristol BS16 1QY UK
Richard.McClatchey@cern.ch
[2]LLP/ESIA, Universit de Savoie
Annecy, 74016 CEDEX, France
Sebastien.Gaspard@cern.ch
[3]ETT and EP Divisions, CERN
Geneva, Switzerland
Jean-Marie.Le.Goff@cern.ch

Abstract. One of the main drivers in object-oriented design for information systems is the need for the reuse of design artifacts in handling systems evolution. To be able to cope with change, systems must have the capability of reuse and to adapt as and when necessary to changes in requirements. To address the issues of reuse in designing evolvable systems, this paper proposes a so-called *description-driven* system architecture. The proposed architecture is based on a two-dimensional design approach founded on the adoption of a multi-layered modeling architecture and on a reflective meta-level architecture. This paper discusses the need for capturing holistic system description when modeling large-scale distributed systems and the role of reflection as a method to cater for reuse in systems evolution. A practical example of the application of this design philosophy, the CRISTAL project, is used to demonstrate the reuse of description-driven data objects to provide for evolution.

1 Introduction

With the advent of the Internet and e-commerce, the need for coexistence and interoperation with legacy systems and for reduced times-to-market the demand for the timely delivery of flexible software has increased. Couple to this the increasing complexity of systems and the requirement for systems to evolve over potentially extended timescales and the importance of clearly defined, extensible models as the basis of rapid systems design becomes a pre-requisite to successful systems implementation.

J.-M. Bruel and Z. Bellahsne (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 101-111, 2002.

Many approaches have been proposed to address aspects of design reuse and implementation for modern object-oriented systems. Each has its merits and focuses on concerns such as data modeling, process modeling, state modeling and lifecycle modeling. More or less successful attempts have been made to combine these approaches into modeling languages or methodologies such as OMT [1] and UML [2] but ultimately these approaches lack cohesion since they are often simply collections of disparate techniques. Recent reports on their use have led to proposals for enhancements such as pUML [3], which have recognized and begun to address these failings.

This paper advocates a design and implementation approach that is holistic in nature, viewing the development of modern object-oriented software from a systems standpoint. The philosophy that has been investigated is based on the systematic capture of the description of systems elements covering multiple views of the system to be designed (including data, process and time views) using common techniques. The approach advocated here has been termed description-driven and its underlying philosophy is the subject of this paper. Essentially the description-driven approach involves identifying and abstracting the crucial elements (such as items, processes, lifecycles, goals, agents and outcomes) in the system under design and creating high-level descriptions of these elements which are stored and managed separately from their instances.

The following section outlines how we have arrived at a description-driven systems philosophy and that is followed by a discussion of the role of reflection in self-description. A practical example of the use of this approach is introduced later in the paper, which is brought to a close with conclusions based on the implications of the use of a description-driven systems design approach.

2 Experiences of Methods, Patterns and Frameworks

Recent experience of implementing complex and dynamic object-oriented systems has indicated that effective software reuse can result from the reuse of high-level design artifacts. One reason for this is that underlying software technology changes so rapidly, this being especially true in software projects that have long timescales, thereby making code reuse difficult. For example object technology has witnessed, in a short space of time an evolution from languages such as Smalltalk, ADA, C++, Java through middleware such as EJB, COM+, Active X and OMG CORBA and the object community is still in a state of flux.

Where we have experienced most success in reuse of software artifacts is with visual modeling languages such as OMT and UML. The creation and evolution of graphical models using UML has allowed us to specify, visualize, construct and document the artifacts of the software systems we have built, adopting an approach in which we concentrate on the descriptive elements of UML. UML has increasingly become the universal object analysis and design modeling standard and as a consequence of this we have over the years been able to reuse large-grained architectural frameworks and patterns which have been captured in UML.

In the object oriented community well-known design patterns [4] are named, described and cataloged for reuse by the community as a whole. We have not only used many well-known patterns but in the domain of description-driven systems

development we have discovered new patterns. It has enabled us to make use of design patterns that were proven on previous projects and is one example of reuse at the larger grain level. UML diagrams are able to describe pattern structure but provide little support for describing pattern behavior or any notation for the pattern template.

Frameworks are reusable semi-complete applications that can be specialized to produce custom applications. They specify reusable architectures for all or part of a system and may include reusable classes, patterns or templates. We have found that frameworks focus on reuse of concrete design algorithms and implementations in a particular programming language, they can be viewed as the reification of families of design patterns and are an important step towards the provision of a truly holistic view of systems design.

Emerging and future information systems however require more powerful data modeling techniques that are sufficiently expressive to capture a broader class of applications. Evidence suggests that the data model must be OO, since that is the model providing most generality. The data model needs to be an open OO model, thereby coping with different domains having different requirements on the data model [5]. We have realised that object *meta-modeling* allows systems to have the ability to model and describe both the static properties of data and their dynamic relationships, and address issues regarding complexity explosion, the need to cope with evolving requirements, and the systematic application of software reuse.

To be able to describe system and data properties, object meta-modeling makes use of meta-data. Meta-data are information defining other data. The judicious use of meta-data can lead to heterogeneous, extensible and open systems [6]. Meta-data make use of a meta-model to describe domains. Our recent research has shown that meta-modeling creates a flexible system offering the following - reusability, complexity handling, version handling, system evolution and inter-operability [7]. Promotion of reuse, separation of design and implementation and reification are some further reasons for using meta-models [8]. As such, meta-modeling is a powerful and useful technique in designing domains and developing dynamic systems.

The use of UML, Patterns and Frameworks as design languages and devices clearly eases difficulties inherent in the timely delivery of large complex object-based systems. However, each approach addresses only part of the overall "design space" and fails to enable a holistic view on the design process. In particular they do not easily model the description aspects or meta-information emerging from systems design. In other words, these approaches can locate individual pieces in the overall design puzzle but do not enable the overall puzzle to be viewed. In the next section we look at reflection as the mechanism to open up the design puzzle.

3 Reflection in Systems Design

A crucial factor in the creation of flexible information systems dealing with changing requirements is the suitability of the underlying technology in allowing the evolution of the system. Exposing the internal system architecture opens up the architecture, consequently allowing application programs to inspect and alter implicit system aspects. These implicit system elements can serve as the basis for change and for extensions to the system. Making these internal structures explicit allows them to be subject to scrutiny and interrogation.

A reflective system utilizes an open architecture where implicit system aspects are reified to become explicit first-class meta-objects [9]. The advantage of reifying system descriptions as objects is that operations can be carried out on them, like composing and editing, storing and retrieving, organizing and reading. Since these meta-objects can represent system descriptions, their manipulation can result in change in the overall system behaviour. As such, reified system descriptions are mechanisms that can lead to dynamically modifiable systems and reusable systems. Meta-objects, as used in the current work, are the self-representations of the system describing how its internal elements can be accessed and manipulated. These self-representations are causally connected to the internal structures they represent i.e. changes to these self-representations immediately affect the underlying system. The ability to dynamically augment, extend and redefine system specifications can result in a considerable improvement in flexibility. This leads to dynamically modifiable systems, which can adapt and cope with evolving requirements.

There are a number of OO design techniques that encourage the design and development of reusable objects. In particular design patterns are useful for creating reusable OO designs [4]. Design patterns for structural, behavioral and architectural modeling have been well documented elsewhere and have provided software engineers with rules and guidelines that they can immediately (re-)use in software development. Reflective architectures that can dynamically adapt to new user requirements by storing descriptive information which can be interpreted at runtime have lead to so-called Adaptive Object Models [10]. These are models that provide meta-information about domains that can be changed on the fly. Such an approach, proposed by Yoder, is very similar to the approach adopted in this paper

A Description-Driven System (DDS) [7] architecture, as advocated in this paper, is an example of a reflective meta-layer (i.e. meta-level and multi-layered) architecture. It makes use of meta-objects to store domain-specific system descriptions, which control and manage the life cycles of meta-object instances, i.e. domain objects. The separation of descriptions from their instances allows them to be specified and managed and to evolve independently and asynchronously. This separation is essential in handling the complexity issues facing many web-computing applications and allows the realization of inter-operability, reusability and system evolution as it gives a clear boundary between the application's basic functionalities from its representations and controls. As objects, reified system descriptions of DDSs can be organized into libraries or frameworks dedicated to the modeling of languages in general, and to customizing its use for specific domains in particular.

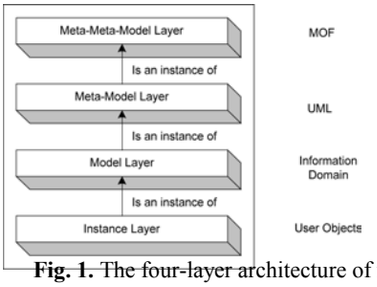


Fig. 1. The four-layer architecture of the OMG

4 Description-Driven Systems

In modeling complex information systems, it has been shown that at least four modeling layers are required [11], see Fig. 1. Each layer provides a service to the layer above it and serves as a client to the layer below it. The meta-meta-model layer defines the language for specifying meta-models. Typically more compact than the meta-model it describes, a meta-meta-model defines a model at a higher level of abstraction than a meta-model. The meta-model layer defines the language for specifying models, a meta-model being an instance of a meta-meta-model. The model layer defines the language for specifying information domains. In this case, a model is an instance of a meta-model. The bottom layer contains user objects and user data, the instance layer describing a specific information domain. The Object Management Group (OMG) [12] standards group has a similar architecture based on model abstraction, with the Meta-Object Facility (MOF) model and the Unified Modeling Language (UML) [2] model defining the language for the meta-meta-model and meta-model layers, respectively.

Orthogonal to the model abstraction inherent in this multi-layered approach is the information abstraction that separates *descriptive* information from the data they are describing. These system descriptions are normally called meta-data, as they are information defining other data. A reflective open architecture typifies this abstraction. A reflective open architecture is divided into two levels - the meta-level where the descriptive information reside and the base level which stores the application data described by the meta-level elements. The meta-level contains the meta-data objects (also referred to as meta-objects in this paper) which hold the meta-data. These meta-objects manage the base level objects.

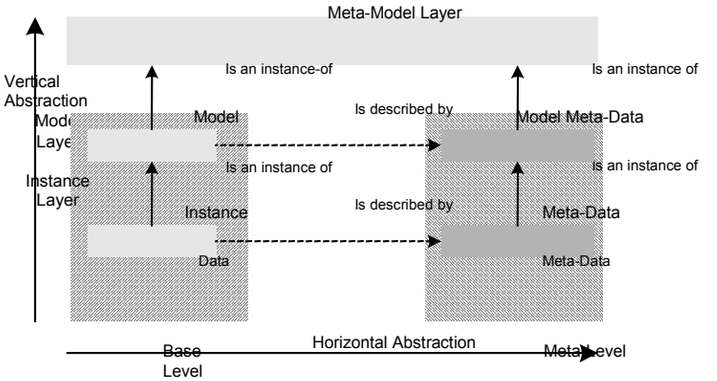


Fig. 2. A Description-Driven Architecture

In a description-driven system as we define it, descriptions are separated from their instances and managed independently to allow the descriptions to be specified and to evolve asynchronously from particular instantiations of those descriptions. Separating descriptions from their instantiations allows new versions of elements (or element descriptions) to coexist with older versions. This paper proposes an architecture that

combines a multi-layered meta-modeling approach with a meta-level architecture [13]. The description-driven architecture is illustrated in Fig. 2.

The layered architecture on the left-hand side of Fig. 2 is typical of layered systems and the multi-layered architecture specification of the OMG (see Fig. 1). The relationship between the layers is *Is an instance of*. The instance layer contains data that are instances of the domain model in the model layer. Similarly, the model layer is an instance of the meta-model layer. On the right hand side of the diagram is another form of model abstraction. It shows the increasing abstraction of information from meta-data to model meta-data, where the relationship between the two is *Is an instance of* as well. These two architectures provide layering and hierarchy based on abstraction of data and information models.

The horizontal view in Fig. 2 provides an alternative abstraction where the relationship of meta-data and the data they describe are made explicit. This view is representative of the information abstraction and the meta-level architecture discussed earlier. The meta-level architecture is a mechanism for relating data to information describing data, where the link between the two is *Is described by*. As a consequence, the dynamic creation and specification of object types is promoted.

The separation of system type descriptions from their instantiations allows the asynchronous specification and evolution of system objects from system types; consequently, descriptions and their instances are managed independently and explicitly. The dynamic configuration (and re-configuration) of data and meta-data is useful for systems whose data requirements are unknown at development time. It is the combination of the *instance of* and *is described by* relationships that provides the holistic approach inherent to description-driven systems.

5 A Relational Description-Driven System

Paper [14] discusses an analogous modeling architecture, which is based on relational models. The horizontal abstraction of the architecture is also based on Instance-of relationship and a meta-modeling approach. Meta-data are generated and extracted from data elements□structures. The data layer corresponds to the instances that are manipulated in the system. The type of meta-data that can be extracted in the data layer concerns the physical aspects of the data such as the volume or localization. The model layer corresponds to concepts for model description describing data and meta-data in the data layer. In relational models, the model layer is the application model. The meta-data that can be extracted from the model layer are descriptive information concerning the physical structure, e.g. data dictionary.

The meta-model layer defines the model formalism that is used in the system. As the architecture is intended for relational systems, the meta-model layer describes the concepts of the relational model, e.g. Table, Relation and Key and the model relating these primitives together. At this level, the type of meta-data that can be extracted concerns tools and methods for inter-operating and relating the different relational models. The meta-meta-model layer is the root modeling level that supports inter-operability and extensibility. The meta-meta-model layer uses conceptual graphs [15] to allow homogeneous representation and manipulation of the lower levels. Conceptual graphs are formalisms where the universe of discourse can be modeled by

concepts and conceptual relations. A relational architecture example is shown in Fig. 3.

Our proposed DDS architecture is similar to the architecture described in [14], but is based on an object model. Both horizontal layerings are based on model abstraction, and meta-data are used as descriptive information to describe the concepts for managing data at each layer. The two architectures are orthogonal in the modeling paradigm used - relational model for [14] and object model for DDS. Thus, the relational architecture uses conceptual graphs and the DDS architecture uses the MOF as its representation formalism in the meta-meta-model layer. In the meta-model layer, the relational architecture uses a relational model and the DDS architecture uses the UML. As a practical example of our approach the next section describes the DDS architecture developed in the context of research carried out in the CRISTAL project at CERN.

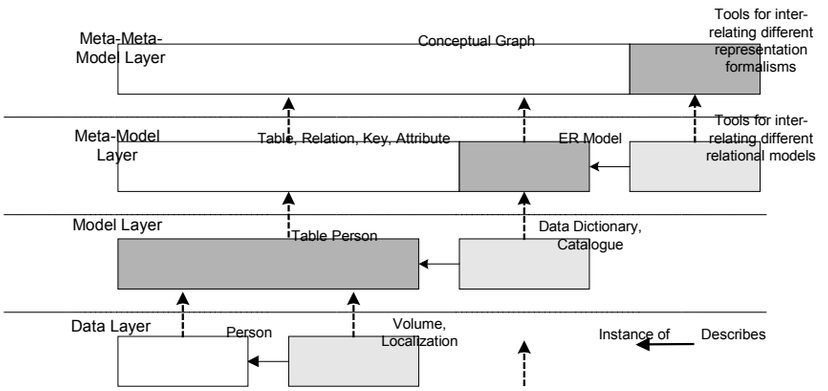


Fig. 3. Relational model represented as a description-driven system

6 An Object-Based Description-Driven Architecture

The Compact Muon Solenoid (CMS) is a general-purpose experiment at CERN [16] that will be constructed from around a million parts and will be produced and assembled in the next decade by specialized centres distributed worldwide. As such, the construction process is very data-intensive, highly distributed and ultimately requires a computer-based system to manage the production and assembly of detector components. In constructing detectors like CMS, scientists require data management systems that are able of cope with complexity, with system evolution over time (primarily as a consequence of changing user requirements and extended development timescales) and with system scalability, distribution and interoperation.

No commercial products provide the capabilities required by CMS. Consequently, a research project, entitled CRISTAL (Cooperating Repositories and an Information System for Tracking Assembly Lifecycles [17]) has been initiated to facilitate the management of the engineering data collected at each stage of production of CMS. CRISTAL is a distributed product data and workflow management system which makes use of an OO database for its repository, a multi-layered architecture for its component abstraction and dynamic object modeling for the design of the objects and

components of the system. CRISTAL is based on a DDS architecture using meta-objects. The DDS approach has been followed to handle the complexity of such a data-intensive system and to provide the flexibility to adapt to the changing scenarios found at CERN which are typical of any research production system. Lack of space prohibits further discussion of CRISTAL; detail can be found in [7] and [13].

The design of the CRISTAL prototype was dictated by the requirements for adaptability over extended timescales, for system evolution, for interoperability, for complexity handling and for reusability. In adopting a description-driven design approach to address these requirements, the separation of object instances from object description instances was needed. This abstraction resulted in the delivery of a three layer description-driven architecture. The model abstraction (of instance layer, model layer, meta-model layer) has been adopted from the OMG MOF specification [12], and the need to provide descriptive information, i.e. meta-data, has been identified to address the issues of adaptability, complexity handling and evolvability.

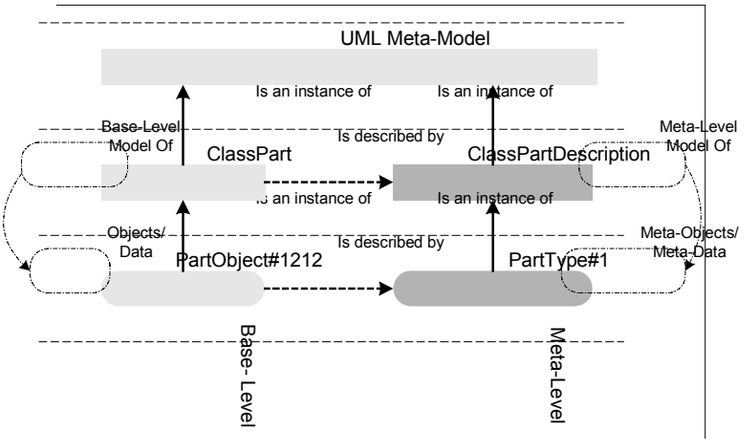


Fig. 4. The CRISTAL architecture

Fig. 4 illustrates the CRISTAL architecture. The CRISTAL model layer is comprised of class specifications for CRISTAL type descriptions (e.g. PartDescription) and class specifications for CRISTAL classes (e.g. Part). The instance layer is comprised of object instances of these classes (e.g. PartType#1 for PartDescription and Part#1212 for Part). The model and instance layer abstraction is based on model abstraction and *Is an instance of* relationship. The abstraction based on meta-data abstraction and *Is described by* relationship leads to two levels - the meta-level and the base level. The meta-level is comprised of meta-objects and the meta-level model that defines them (e.g. PartDescription is the meta-level model of PartType#1 meta-object). The base level is comprised of base objects and the base level model which defines them (Part is the base-level model of Part#1212 object).

In the CMS experiment, production models change over time. Detector parts of different model versions must be handled over time and coexist with other parts of different model versions. Separating details of model types from the details of single parts allows the model type versions to be specified and managed independently,

asynchronously and explicitly from single parts. Moreover, in capturing descriptions separate from their instantiations, system evolution can be catered for while production is underway and therefore provide continuity in the production process and for design changes to be reflected quickly into production. The approach of reifying a set of simple design patterns as the basis of the description-driven architecture for CRISTAL has provided the capability of catering for the evolution of a rapidly changing research data model. In the two years of operation of CRISTAL it has gathered over 20 Gbytes of data and been able to cope with 25 evolutions of the underlying data schema without code or schema recompilations.

7 Conclusions

Reflection provides a sound foundation for the specification of meta-level architectures and gives the capability of customizing system behavior through explicit descriptive mechanisms of implicit system aspects. The transformation of implicit system aspects to explicit description meta-objects is termed reification. These description meta-objects comprise the meta-layer that manages and controls the life cycles of the base-layer objects it describes. By reifying system descriptions as meta-objects, they can be treated, accessed and altered as objects. As a consequence, this work has demonstrated that through reification, system behavior can be accessed and manipulated at runtime thereby creating a dynamically modifiable system.

The combination of a multi-layered meta-modeling architecture and a reflective meta-level architecture resulted in what has been referred to in this paper as a description-driven systems (DDS) architecture. A DDS architecture, is an example of a reflective meta-layer architecture. The CRISTAL DDS architecture was shown to have two abstractions. The vertical abstraction is based on the OMG meta-modeling standard, and has three layers - instance layer, model layer and meta-model layer. This paper has proposed an orthogonal horizontal abstraction mechanism that complements this OMG approach. The horizontal abstraction is based on the meta-level architecture approach, and has two layers - meta-level and base level. The relationship between the vertical layers is Instance-of and the relationship between the horizontal layers is Describes.

The approach taken in this paper is consistent with the OMG's goal of providing reusable, easily used and integrated, scalable and extensible components [18]. Likewise, the contributions of this work complement the ongoing research on Adaptive Object Model (AOM) espoused in [19], where a system with an AOM (also called a Dynamic Object Model) is stated to have an explicit object model that is stored in the database, and interpreted at runtime. Objects are generated dynamically from the AOM schema meta-data that represent data descriptions. The AOM approach also uses reflection in reifying implicit data aspects (e.g. database schema, data structures, maps of layouts of data objects, references to methods or code).

The description-driven philosophy facilitated the design and implementation of the CRISTAL project with mechanisms for handling and managing reuse in its evolving system requirements. In conclusion, it is interesting to note that the OMG has recently announced the so-called Model Driven Architecture as the generic basis of future systems integration [18]. Such a philosophy is directly equivalent to that expounded in this and earlier papers on the CRISTAL description-driven architecture.

Acknowledgments

The authors take this opportunity to acknowledge the support of their home institutes and numerous colleagues responsible for the CRISTAL software.

References

1. Rumbaugh J. et al., Object-Oriented Modeling & Design Prentice Hall (1991)
2. The Unified Modeling Language (UML) Specification, URL http://www.omg.org/technology/uml/.
3. pUML Initial Submission to OMG's RFP for UML 2.0 Infrastructure. URL http://www.cs.york.ac.uk/puml/.
4. Gamma E., Helm R., Johnson R. and Vlissides J., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.
5. Klas W. and Schrefl M., "Metaclasses and their Application. Data Model Tailoring and Database Integration", Lecture Notes in Computer Science 943. Springer. 1995.
6. Crawley S., et. al., "Meta Information Management", Proceedings of the Second IFIP International Conference on Formal Methods for Open Object-based Distributed Systems, Canterbury, United Kingdom, July 1997.
7. Kovacs Z., "The Integration of Product Data with Workflow Management Systems", PhD Thesis, University of West of England, Bristol, England, April 1999.
8. Blaha M. and Premerlani W., "Object-Oriented Modeling and Design for Database Applications", Prentice Hall, 1998.
9. Kiczales G., "Metaobject Protocols: Why We Want Them and What Else Can They Do?", Chapter in Object-Oriented Programming: The CLOS Perspective, pp 101-118, MIT Press, 1993.
10. Foote B. and Yoder J. "Meta-data and Active Object-Models". Proc. of the Int. Conference on Pattern Languages Of Programs, Monticello, Illinois, USA, August 1998.
11. Saudt M., Vaduva A. and Vetterli T., "Metadata Management and Data Warehousing", Technical Report 21, Swiss Life, Information Systems Research, July 1999.
12. The Object Management Group (OMG), URL http://www.omg.org.
13. Estrella F., "Objects, Patterns and Descriptions in Data Management", PhD Thesis, Uni-versity of the West of England, Bristol, England, December 2000.
14. Kerherve B. and Gerbe O., "Models for Metadata or Metamodels for Data", Proceedings of the Second IEEE Metadata Conference, Maryland, USA, September, 1997.
15. Sowa J., "Conceptual Structures: Information Processing in Mind and Machine", Addison-Wesley, 1984.
16. The European Centre for Nuclear Research (CERN), URL http://cern.web.cern.ch/CERN.

17. Estrella F. et al., "Handling Evolving Data Through the Use of a Description Driven Sys-tems Architecture". Lecture Notes in Computer Science Vol 1727, pp 1-11 ISBN 3-540-66653-2 Springer-Verlag, 1999.
18. OMG Publications., "Model Driven Architectures - The Architecture of Choice for a Changing World". See http://www.omg.org/mda/index.htm
19. Yoder J., Balaguer F. & Johnson R., "Architecture and Design of Adaptive Object-Models". Proc of OOPSLA 2001, Intriguing Technology Talk, Tampa, Florida. October 2001.

A Specification-Oriented Framework for Information System User Interfaces

Eliezer Kantorowitz and Sally Tadmor

Computer Science Department
Technion Israel Institute of Technology
32000 Haifa, Israel
{kantor,sally}@cs.technion.ac.il
http://www.cs.technion.ac.il/~kantor/

Abstract. A costly part of software development regards verification, i.e., checking that the code implements the specification correctly. We introduce the concept of *specification-oriented* frameworks, with the purpose of facilitating verification. A specification-oriented framework enables direct translation of the specifications into code, whose equivalence with the specification is easy to establish. The feasibility of this concept was investigated with the experimental *Simple Interfacing* (SI) framework for construction of user interface software of interactive information systems. Such an information system is constructed by translating the natural-language use-case specification into SI based code. SI provides high-level methods for data entry and data display. The use-case coding assumes that a database schema and required complex data manipulations are developed separately. The code produced for the use-cases of five small projects corresponded quite closely to the natural-language specifications and facilitated considerably the verification. Further research is required to assess the usefulness of the approach.

1 Introduction

The software engineering challenge regards the manufacturing and management of complex systems. One mean to overcome these complexities is the use of high-level tools such as very high-level languages and frameworks of high-level software components. This investigation is concerned with the design of frameworks for construction of graphical user interfaces (GUI) for interactive information systems (IS). It considers frameworks for use-case oriented software development processes, such as the Unified Software Development Process (USDP) [1]. The USDP was designed in connection with design of the Unified Modeling language (UML) [2]. We are especially interested in use-case oriented processes as they facilitate the manufacturing of systems with high usability levels, i.e., systems that enable the users to accomplish all required tasks with a minimum of effort and in a pleasant way. This is achieved by employing usability considerations in the design of the use-cases. The USDP process

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 112-121, 2002.

begins with requirement elicitation and continues with the specification of the system by its use-cases. First a natural-language use-case specification is developed and its usability is validated. The use of natural language enables validation by domain experts that are not familiar with formal specification methods. The validated use-cases are then translated into formal UML use-case diagrams. The process continues with the analysis of the formal use-case specification, system design, implementation and testing. The testing phase includes *verification*, i.e., checking that the code implements the use-case specification correctly. This is usually done by testing each one of the different use-cases with sets of test data that cover the different kinds of possible scenarios (black box testing). The verification may also be accomplished with formal methods that assume that the natural-language use-case specification has been correctly translated into a formal use-case specification. Using formal methods, e.g. [3], to show that the code implements the formal specification is not a trivial task. The verification process involves a considerable effort. In some situations it is possible to produce the code automatically from the specifications and thus completely avoid the costly verification. One example is a statecharts specification of reactive systems [4] that may be translated automatically to code [5]. We are, however, not aware of a general method for producing the code of information systems from their specification. This study suggests therefore a less ambitious approach to reduce the verification effort.

In the approach suggested in this paper, the natural-language use-case specification is translated into high-level code, which implements the use-cases. The verification effort in this approach is reduced to showing the equivalence between the natural-language use-case specification and the code. We denote a framework that enables a direct coding of the specifications as a *specification-oriented framework*. The process proposed in this paper differs from the USDP process where the use-cases are not coded, but form the basis for the analysis and design of the system. The feasibility of the approach proposed in his paper depends on whether it is possible to design an appropriate high-level language or component framework for coding of use-case activities. In this research we investigate the feasibility of a use-case specification-oriented framework for construction of the GUI software of interactive information systems. To the best of our knowledge, none of the frameworks existing today has been especially designed to support use-case specification-oriented system development.

2 Model of the Experimental System

This section describes our use-case oriented model of the GUI software of interactive information systems. Later we describe our experimental implementation of this model, using our Simple Interfacing (SI) framework. Our model suggests IS software to be composed of four parts. The first part is the implementation of the use cases, based on some basic actions. The second part is a framework implementing these basic actions. The third part is a database and the last part is data manipulation software.

In SI we employ the following basic use case actions:

1. displaying data to the user,
2. getting data from the user
3. getting data from the database
4. inserting and updating data in the database
5. general purpose data manipulation

By database we mean a persistent database that may be employed for sharing of data between different applications. Our five basic actions hide the geometrical properties of the Graphical User Interface (GUI), its related event handling and the database access methods. The basic actions of our model only specify the flow of data between the user, the system and the database. In other words, they specify the input and output of data to the system. Our basic actions do not specify how the in/output is done. The motivation for selecting this abstraction is that we consider the input and output of data to the system to be the semantics of the user-interface system. The purpose of the graphical layout of the user-interface is to facilitate the work of the user.

General-purpose data manipulation is done by tools such as the SQL language. However, a use case specification may beyond the above five basic operations employ special kinds of data manipulation, e.g. scheduling tennis games by special tournament rules, finding the least expensive air line connection between two cities. Such special data manipulation operations cannot be part of our general purpose SI use case framework. It is therefore assumed that frameworks for such special operations are developed separately. A use case is coded by combining the classes of the general-purpose framework with the classes of operations that are specific to the application.

The design of the experimental SI system is described in Appendix 1. An example of the use of SI is shown in Appendix 2. The next section evaluates the results of five small SI based projects.

3 Evaluation

3.1 Code is Data-Centered

Data input and output are considered to be the basic operators of the user interface. SI based code is basically data-centered. The I/O operators of SI are implemented by methods of the `PaneController` class listed in Tab. 1 and Tab. 2.

Table 1. Input from the user and output to her/him

Input methods	Output methods
`RequestInput`	`Display`
`GetInput`	`AddLabel`
`AddButton`	`AddPicture`

Table 2. Input to the database and output from it

Input methods	Output methods
`Update`	`Display`

3.2 Ease of Verification and Traceability

In the tested examples, the use of the SI framework produced code that resembles the natural-language use-case specification. This facilitated the verification considerably. We have examined the verification of the use-case *Select Offer* (Appendix 2). We compared the above SI based implementation with another implementation employing only Java and its Swing and JDBC packages. The SI based code hides the GUI and database access details and is therefore shorter and seemingly easier to understand. As shown in Table 2, the SI based code is only 61 lines long, while the code using the Swing and JDBC is 147 lines long. Another expression of the shortness of the SI based code is that each statement in the natural-language use-case specification translates on the average into 2.5 lines of SI based code [6], while it takes 7 lines of Java code using the Swing and JDBC packages.

The verification of the SI based code was further facilitated by the observed ease of tracing of the lines of code that corresponds to each statement in the natural-language use-case specification [6]. This ease of tracing is because the code of a particular use-case specification is found in a class that has the same name as the use case and extends the SI class `UseCase`. Tracing the code that implements a use-case specification in the longer non-SI based Java code was noticeable more difficult. The ease by which the code that implements a use case can be traced in SI based code is also expected to facilitate future modifications of the use-cases. However, verifying specifications of the user-interface appearance requires familiarity with the interaction-styles of SI, since the SI based code hide these details.

Table 3. Length of code to implement a use-case, with SI and with Swing and JDBC

	Total number of lines of the use-case class code	Average number of lines of code implementing a single statement in the natural-language use-case specification
Implementation with Java, Swing and JDBC	147	7
Implementation with Java and SI	61	2.5

3.3 Learning Time and Interaction-Styles Limiting

Coding with SI requires understanding the use of the underlying interaction styles, which requires some learning. These standard interaction styles were satisfactory in four of the five tested projects. In one project the programmers wanted a different GUI appearance, e.g., using different widgets, and placing them differently. This required developing a new interaction style. Writing a new style requires understanding of the internal structure of SI, i.e., the methods of the `StyledPane` and `StyledFrame` classes, inheriting from one or two of these classes, and overriding some of their methods. Reaching this higher level of understanding required further learning.

4 Conclusions and Further Research

The use of SI produced code that is relatively easy to verify. It is also expected to facilitate future use case modifications. The use of SI also reduced the coding effort. In one example the code written with SI had only 40% of the length of equivalent Java code exploiting the Swing and JDBC packages. Using the ready-made interaction-styles standardizes the appearance of the GUI.

It is interesting to compare the specifications-oriented SI package with the GUI-oriented Swing package of Java. Swing has classes for detailed construction of GUI widgets. Input and output with these Swing widgets requires detailed coding. SI is on the other hand centered on abstract input and output operations. The SI widgets are also at a high level of abstraction hiding the geometrical details in interaction styles.

The interaction-styles provided by the current version of SI were not satisfactory in one of the five test projects. The design of an interaction-style involves a difficult tradeoff between an interaction-style being general-purpose and being able to meet special requests. More research on design of interaction-styles is needed.

SI has been tested with small systems. To fully assess the approach it must be tried on large systems. Considering the quite promising results, we believe the approach deserves further investigations.

References

1. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process, Addison-Wesley, (1999).
2. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, Addison-Wesley, (1999).
3. Chechik, M., Gannon, J.: Automatic Analysis of Consistency between Requirements and Designs, IEEE Transactions on Software Engineering, 27,7,(July 2001).
4. Harel, D.: Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming, 8,(1987) 231-174, North Holland.
5. Rhapsody product of I-Logix company, see http://www.ilogix.com/ products/rhapsody/index.cfm
6. Tadmor, S.: A Framework for Interactive Information Systems, An M. S. thesis, Technion Institute of Science, Israel, (2002).
7. Goldberg, A. J., Robson, D.: SmallTalk-80: The Language and Its Implementation, Addison-Wesley, (1983).
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Addison-Wesley, (1995).
9. Kantorowitz, E., Sudarsky, O.: "The Adaptable User-interface", Commun.ACM, 32, 11,(Nov 1989),1352-1352.

Appendix 1. Structure of the Experimental SI Framework

A1.1 Design Decisions for SI

This section discusses the design decisions made for the experimental SI system:

1. Implementing SI with Java and its Swing and JDBC (Java Data Base Connectivity) packages. These tools are considered to be * the state of the art and yet sufficient mature.
2. Using a relational database and SQL, powerful and widely available technologies.
3. Using the Model-View-Controller (MVC) design pattern [7] [8] for the design of SI. Using the MVC pattern is expected to facilitate modifications that may be needed at later stages
4. Using a use-case as an implementation unit. Each use-case is implemented separately, by one Java class, which extends the SI `UseCase` class. In terms of MVC the use-case class is the `Controller`. The schema of the relational database may implement the Model of the MVC pattern.
5. Separating the specification of the *interaction-style,* i.e., the way the GUI appears, from the systems functionality. This will enable the interaction-styles to be reused in different applications. From the MVC point of view, the interaction-style is part of the View.

The idea of interaction-styles is inspired by the *dialogue modes* [9]. We define an *interaction-style* of a GUI as the set of all its properties, e.g., colors of widgets, sizes, layout of widgets on the pane, and also the types of widget used. Once constructed, an interaction-style may be employed in a number of different information systems. SI comes with a number of ready-made interaction-styles. The implementer of an information system may, however, define additional reusable interaction-styles to meet special needs.

A1.2 Using SI

The manufacturing of an information system with the SI framework begins with the usual requirement elicitation and detailed natural-language specification of the use-cases. The process continues with an analysis resulting a schema of the system's database. The analysis and design of database schema is, however, not needed in the many cases, where it is required to employ an existing database of a given enterprise. The system may now be implemented by writing a use-case class for each one of the specified use-cases. By convention we name this class with the name of the use-case that it implements. This code is essentially a translation of the statements of the natural-language use-case specification into appropriate method calls. These methods belong either to classes of the SI framework or to classes provided by the programmer for application specific manipulations. Each statement in the natural-language use-case specification is typically translated into one or two method calls. The verification of the resulting code is done by checking the equivalence between this code and the

natural-language use-case specification. The database is created separately using a relational database system.

The following are the five service classes of SI that are visible to the programmer. Their important methods are explained in the following sections.

1. `UseCase` – all use-case classes inherit from this class. This class has a documentation role: since all use-case classes inherit from it, it is clear what the use-case classes of the system are. Apart from this role, this class gives some hidden flow-of-control services.
2. `FrameController` – the code of the use-case classes use this class to build *frames* of the GUI. A *frame* is the basic GUI unit defined by the Java Swing package. Frames have Swing *panes* added to them.
3. `PaneController` – the code of the use-case classes use this class to build panes, and attach them to frames of the GUI.
4. `DBProxy` – gives database services to all framework and system classes. System classes use this class when the database services of the `PaneController` class are not sufficient.
5. `Constants` – keeps the constants of the system.

The above few classes and the methods that are listed in the next section are what a use-case programmer has to learn. According to our experiments [6], it takes 4 hours. When the interaction-styles that come with **`SI`** are not satisfactory, new interaction-styles must be implemented. Learning to do this took 4 more hours

GUI Construction. In SI panes and frames are implemented by instances of the `PaneController` and `FrameController` classes. A singular human-computer interaction is in SI handled with one particular pane. The `PaneControllers` give all the services of the basic actions: displaying data to the user, getting data from the user, and updating the database.

The GUI is designed according to the MVC model [8]. The `PaneController` and the `FrameController` are the Controllers of the GUI. Two classes, called the `StyledPane` and `StyledFrame`, implement the View and Model.

The services of the `FrameController` class are:

1. `addPane(PaneController pane)` – adds a pane to the frame.
2. `finallySet()` – sets the frame to be visible and packs it.
3. `close()` – closes the frame.

The services of the `PaneController` class are:

1. `display(String query)` – executes the query by accessing the database, and displays the resulting relation on the pane, in a few text-boxes or in a table. The appearance of the pane is determined by the selected interaction-style.
2. `requestInput(String[] SQLTypes, String[] captions)` – adds widgets to the pane, to request these input data types. The widgets have these captions.

3. `getInput()` – after requesting input from the user, this methods gets the data inserted by the user in the widgets of the pane. The method returns a vector of objects, containing the data inserted by the user.
4. `update(String SQLUpdateQuery)` – executes the query on the database.
5. `addButton(String method, String caption)` - adds a button to the pane, with this caption. When the button is pressed, this method is called.
6. `addButton(String caption)` - adds a button to the pane. Pressing this button will call no method. This method is used for prototyping.
7. `addLabel(String caption)` – adds a label with this caption to the pane.
8. `addPicture(String fileName)` – adds the picture in this file to the pane.

Connecting to the Database. A system can connect to the database using the `DBProxy` class. The services of this class are:

1. `ResultSet executeQuery(String query)` – executes the query, returning a result set.
2. `executeUpdate(String query)` - executes the update query.
3. `close()` - closes the connection to the database.

Flow of Control Services. These services enable a use-case to return control to the calling use-case or other use-cases. These services are given by the UseCase class, and explained in [6].

Appendix 2. Example – SelectOffer Use Case

In this section we bring a short example, to demonstrate the quality of code written using SI. We take the *Select Offer* use-case of the *Offers* system [6]. The use-case natural-language specification is:

"The user sees a frame with two lists, a list of cities and a list of items. The user selects a city and an item, and can press one of two buttons: the Cancel button, or the See Suppliers button. If the user presses the Cancel button, the frame is closed. If the user presses the See Suppliers button, a list of suppliers appears..".

By analyzing all the use cases of the application, the schema of the relational data employed by the code below was designed. The analysis and design of this schema was done traditionally. The resulting SI based code of the use case appears on the next page. It produces the frame shown in Fig. 1.

```
1. import java.util.*;
2. import si.*;
3.
4. public class SelectOffer extends UseCase{
5.
6. private String customerID = null;
7.
8. private FrameController frame = null;
9.
10.   private PaneControllerListStyleWidth2 pane =
   null;
11.
12.   public SelectOffer(String customerID){
13.     super();
14.     frame = new FrameController("What offers
   are you interested in?");
15.     pane = new PaneControllerListStyle-
   Width2(this);
16.     this.customerID = customerID;
17.     pane.addLabel("Select city:");
18.     pane.display("select distinct city from
   Supplier");
19.     pane.addLabel("Select item:");
20.     pane.display("select distinct name from
   Item");
21.     pane.addButton("seeSuppliers","See Suppli-
   ers");
22.     pane.addButton("cancel","Cancel");
23.     frame.addPane(pane);
24.     frame.finallySet();
25.   }
26.
27.   public void seeSuppliers(){
28.       //…
29.   }
30.
31.   public void cancel(){
32.       //…
33.   }
34.
35. }
```

Fig.1. The frame displayed by the *Select Offers* use-case

The Role of Pattern Languages in the Instantiation of Object-Oriented Frameworks

Rosana T. V. Braga * and Paulo Cesar Masiero **

Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo, Brazil
{rtvb,masiero}@icmc.sc.usp.br

Abstract. In this paper we propose the use of pattern languages to guide an object-oriented framework instantiation. Both the framework and the pattern language refer to the same domain, and the framework must have been constructed based on the pattern language. The framework instantiation here proposed is done in several steps, all of them supported by the pattern language. This makes it easier for the developer to instantiate applications, as the knowledge about the pattern language is used during the instantiation process. The proposed approach is illustrated with the example of a framework we have built based on a pattern language for an information systems' domain.

Keywords: Software reuse, frameworks, pattern languages, framework instantiation.

1 Introduction

Software patterns and pattern languages aim at reuse in high abstraction levels. Software patterns try to capture the experience acquired during software development and synthesize it in a problem/solution form [1]. A pattern language is a structured collection of patterns that organize the knowledge about a specific domain. Its patterns can be applied systematically and represent the temporal sequence of decisions that lead to the complete design of an application, so it becomes a method to guide the development process [2].

Object-oriented software frameworks (from now on called simply frameworks) are a set of abstract and concrete classes that can be customized to specific applications, allowing the reuse of large software structures in a particular domain [3]. Families of similar but non-identical applications can be derived from a single framework. However, frameworks are often very complex to build, understand, and use. Framework instantiation, which consists of adapting the framework to the specific application requirements, is complex and, most times, requires a complete understanding of the framework design and implementation details. According to Fayad and Johnson [4], the time to learn a framework to begin

* Financial support from FAPESP Process n. 98/13588-4.
** Financial support from FAPESP and CNPq

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 122–131, 2002.

using it can vary from one to one hundred days, depending on the framework size and comprehensiveness. This factor definitely impacts the final cost of the application and, thus, can inhibit the use of this technology.

Pattern languages can be used together with frameworks to enhance software reuse, as both of them embody experience in specific domains. A pattern language can be used for documenting the framework, as already shown in several works [5, 6]; for supporting the framework design and implementation [2]; and as a method to guide the transformation of the framework in a concrete application [2]. Thus, the availability of a pattern language for a specific domain and its corresponding framework imply that new applications do not need to be built from scratch, because the framework offers the reusable implementations of each pattern of the pattern language.

Several approaches have been proposed to help instantiating frameworks [7, 8, 9]. They rely on the framework documentation to obtain the information needed to instantiate an application, which can include the framework class hierarchy, examples of applications derived from the framework, or cookbooks that show how to adapt the framework to a particular application. Also, patterns can be used to document the framework and to show how to use it [6, 2].

Although several researchers have noticed the relationship between pattern languages and frameworks, no work exists, to the author's knowledge, showing how to use a pattern language in the whole process from framework construction to framework instantiation. This paper explores this idea and shows how to support framework instantiation based on pattern languages, making the instantiation process easier and more systematic. The proposed process consists of four steps, explained in detail in sections 3 to 6: system analysis, mapping between analysis model and framework, implementation of specific classes, and test of the resulting system. The framework must have been built based on the same pattern language using the general process we have described in a previous work [10], not shown here due to space restrictions.

The paper is organized as follows. Section 2 shows an example of a pattern language and its associated framework. Section 3 presents the first step of the instantiation, which consists of the system analysis guided by the pattern language. Section 4 shows how to map the resulting analysis model to the framework. Section 5 gives some advices about the implementation of the specialized classes, as our proposal is aimed at white-box framework instantiation. Section 6 shows the resulting application testing phase. Section 7 presents the concluding remarks.

2 Example of a Pattern Language and Its Associated Framework

To illustrate our approach, we use the business resource management domain, for which we have built a pattern language, denominated GRN[11], a white-box framework, denominated GREN[12], and a tool to automatize the instantiation process. In this paper we use only the GREN white-box version.

Table 1. Example of the GREN documentation

Pattern	Variant	Pattern class	GREN class
1 - Identify the Resource	All	Resource	Resource
	Default, Multiple types	Resource Type	SimpleType
	Nested types	Resource Type	NestedType
9 - Maintain the Resource	All	Resource Maintenance	ResourceMaintenance
		Source Party	SourceParty
		Destination-Party	DestinationParty

The GRN pattern language was built based on the experience acquired during development of systems for business resource management. Business resources are assets or services managed by specific applications, as for example videotapes, products or physician time. Business resource management applications include those for rental, trade or maintenance of assets or services. GRN has fifteen patterns that guide the developer during the analysis of systems for this domain. There are patterns concerning the identification, quantification and storage of the business resource; to deal with several types of management that can be done with business resources, as for example, rental, reservation, trade, quotation, and maintenance; and to treat details common to all types of transactions, as for example payment and commissions.

Figure 1 shows the MAINTAIN THE RESOURCE pattern, extracted from GRN [11]. Observe that the pattern structure diagram uses the UML notation with some modifications. We have included special markers before input and output system operations ("?" and "!", respectively). Furthermore, a "&" before a method name means that its call message is sent to a collection of objects.

The GREN framework was developed to support the development of applications modeled using the GRN pattern language, i.e., it implements all the behavior provided by classes, relationships, attributes, methods, and operations of GRN patterns. It was programmed using VisualWorks Smalltalk (from Cincom Systems) and the MySQL database, for object persistence. Its first version contains about 150 classes and 30k lines of code. Its documentation provides several tables, built based on GRN, to guide its instantiation for specific applications. For example, Table 1 is used in the creation of the concrete classes of the application layer (there is another similar table for the graphical user interface layer, not shown here). Other tables list the methods to be overridden in the newly created classes. These methods are defined in GREN abstract classes and need to be overridden by their subclasses.

3 System Analysis

The first step of the instantiation process is the analysis of the specific system to be developed using the framework. Based on the specific system requirements, the pattern language is used to model the system. The pattern language should have been studied in advance by the developer to determine if it can be applied in the analysis of this specific system. It contains analysis patterns, each of which

Pattern 9: MAINTAIN THE RESOURCE

Context

Your application deals with resource maintenance or repair. You have already identified and quantified these resources, which are basically customer assets that present faults or need periodic maintenance. . .

Problem

How do you manage resource maintenance performed by your application?

Forces

- Keeping maintenance records is important both to customers and to organizations that do maintenance. . .

Structure

Figure 18 shows the MAINTAIN THE RESOURCE pattern.

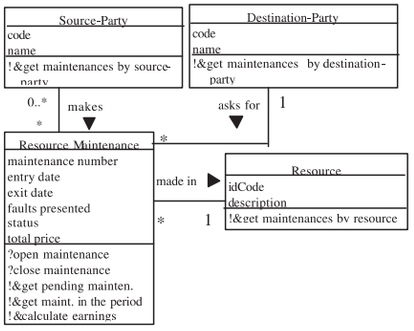


Figure 18: MAINTAIN THE RESOURCE pattern

Participants

Resource Maintenance : represents all the details involved in maintaining a resource. The attribute faults presented describes what is wrong with the resource. The method open maintenance is used to register the resource to be repaired; close maintenance is executed when the maintenance finishes, and get pending maintenances retrieves maintenances that are not finished. The attribute status denotes the maintenance stage: pending, partially fulfilled, or fully fulfilled.

Resource: as described in previous patterns.

Source-Party: represents the department or branch of the organization that is responsible for doing the maintenance. This class is optional in this pattern because in small organizations, with no branches or departments, it is not worth creating it.

Destination-Party: represents the owner of the resource being maintained as, for example, the customer.

Example

. . .

Following patterns

Check now the other patterns in Section 2.3, which deal with other maintenance details After, check the convenience of using the QUOTE THE MAINTENANCE (10) and the IDENTIFY MAINTENANCE TASKS (14) patterns.

Fig. 1. Part of maintain the resource pattern

containing at least the usual elements of a pattern, like problem, context, forces, and solution. In particular, the solution contains a class diagram illustrating the pattern participants. It is also desirable to have a diagram showing the interaction among the patterns of the pattern language.

A pattern language usually has all the elements needed for its usage: a context section, where there are elements that should be part of an application context in order to use the pattern; a problem statement that should be checked to see if it matches the problem to be solved; some forces that reflect the restrictions, benefits and drawbacks confronted in the system; a solution and possible variants or sub-patterns that help finding the best solution to the problem. Other elements such as consequences, examples, known uses, etc. can be present in the patterns. As each pattern is applied, it is recommended to take notes of the roles played by each class, of the variants or sub-patterns used, and of the additional attributes and methods. This will constitute the “history of patterns

and variants applied", which is useful in the future framework instantiation. A small class diagram is produced after the application of the first pattern. This diagram grows as new patterns are applied, and turns into the class diagram of the specific system.

After having applied the pattern language, the requirements document is checked to find non-attended or partially attended requirements. The class diagram is complemented to fulfill them, by adding new attributes, classes, relationships, methods and operations. These complements should be highlighted to distinguish them from the rest of the diagram. Also, notes should be taken to document the analysis decisions made here, because they are essential in the future maintenance of the system. The requirements document should be annotated to discern requirements not covered by the pattern language, as this can be used to improve the pattern language.

The example used in this paper to illustrate our approach is of a Pothole Tracking Repair System (PHTRS), whose requirements were established by Pressman [13] and are reproduced here. "Citizens can log onto a Web site and report the location and severity of potholes. As potholes are reported they are logged within a "public works department repair system" and are assigned an identifying number, stored by street address, size (on a scale of 1 to 10), location (middle, curb, etc.), district (determined from street address), and repair priority (determined from the size of the pothole), Work order data are associated with each porthole and includes pothole location and size, repair crew identifying number, number of people on crew, equipment assigned, hours applied to repair, hole status (work in progress, repaired, not repaired), amount of filler material used and cost of repair (computed from hours applied, number of people, material and equipment used). Finally, a damage file is created to hold information about reported damage due to the pothole and includes citizen's name, address, phone number, type of damage, dollar amount of damage. PHTRS is an on-line system. All queries are to be made interactively."

PHTRS requirements were analyzed using GRN, producing its analysis model, shown in Figure 2. Tags show the role played by the class it points to. Their format is "P#n: role", where "n" is the pattern number and "role" is the role played by the class in that pattern. Notice that we do not represent canonic methods and operations. During the analysis, a match has to be done between the pattern attributes, methods and operations versus the concrete class attributes, methods and operations. For the PHTRS example, some additional attributes were added, as for example the attribute `numberOfPeopleOnCrew` (`WorkOrder` class).

Table 2 shows some of the seven patterns used during the instantiation, together with the roles played by each application class. During the application of pattern 1 - Identify the Resource, Pothole was the resource managed by the application. It was considered as a multiple-typed resource, because we may need to classify potholes by district, by size, by location or by citizen. When applying pattern 2 - Quantify the Resource, we decided that a pothole is unique, so we have chosen the Single Resource sub-pattern (other options, like

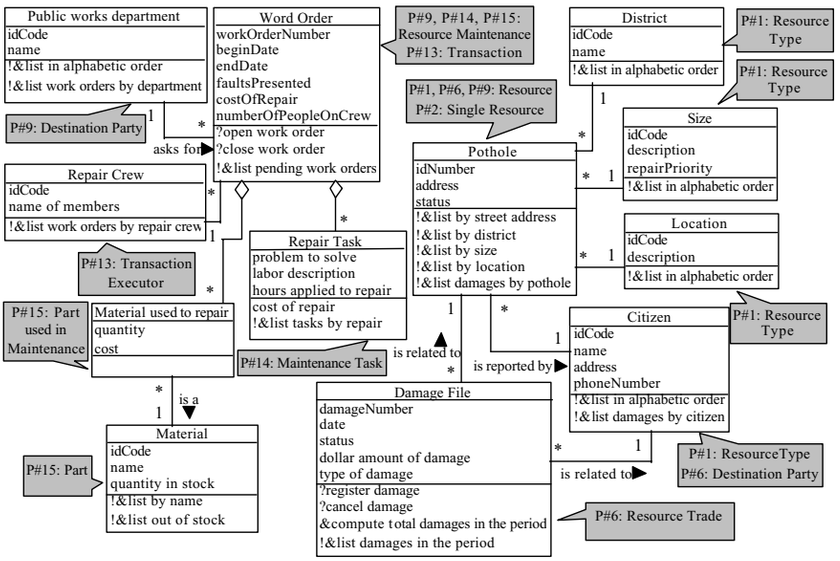


Fig. 2. PHTRS analysis model with patterns

resources that are measured, instantiated or dealt with by lots were not adequate in our case). After having skipped some patterns not concerned with PHTRS, we have applied pattern 9 - Maintain the Resource, pattern 13 - Identify the Transaction Executor, pattern 14 - Identify the Maintenance Tasks, and pattern 15 - Identify the Maintenance Parts.

At this point, the only requirement not attended by GREN was the damage file creation. If we analyze the semantic of GRN patterns, we do not find a pattern to address this requirement, as we want to log the damages caused to citizens due to the pothole, and this is not a rental, trade or maintenance transaction.

Table 2. History of patterns used in the instantiation

Pattern	Variant	Participant	Application Class
1 - Identify the Resource	Multiple types	Resource	Pothole
		Resource Type	District
		Resource Type	Size
		Resource Type	Location
		Resource Type	Citizen
2 - Quantify the Resource	Single Resource	Resource	Pothole
9 - Maintain the Resource	No source party	Resource	Pothole
		Resource Maintenance	Work Order
		Destination-Party	Public works department
. . .			
6 - Trade/Use the Resource	No source party	Resource	Pothole
		Resource Trade	Damage File
		Destination-Party	Citizen

However, if we analyze the patterns' syntax, we see that the Resource Trade has attributes similar to those we are looking for to our damage file. So, we can use pattern 6 - Trade the Resource, to model the damage file. This enabled us to use GREN in a way not envisioned by its designers, i.e., the pattern has the required computational structure, although it does not carry the domain semantics anticipated by the designer. We call this "semantic/ syntatic replacement process", and note it in Table 2 by the general Use the Resource pattern.

The result of this step is the PHTRS analysis model (Figure 2) and the history of patterns used in the instantiation (Table 2). We did not produce a list of analysis decisions made, because GREN supported all the functionality needed for PHTRS. This list would be required if we had included additional behavior, not covered by GREN, to be manually programmed in the implementation step.

4 Mapping between the Analysis Model and the Framework

The second step of the instantiation process is to map the resulting analysis model to the framework. We consider that the framework was documented according to the guidelines suggested in our process for building a framework based on a pattern language [10], i.e., the relationship between the patterns of the pattern language and the framework classes is adequately documented. Considering the "history of patterns and variants applied", created during system analysis, the framework documentation can be used to identify the classes and methods that need to be created to adapt the framework according to the patterns and variants applied.

As an example, the mapping between PHTRS and GREN was done using the special documentation provided by GREN. As mentioned in Section 2, this documentation consists of several tables showing, for each GRN pattern, the corresponding GREN classes to inherit from and the methods to be overridden during instantiation. For example, based on Table 2, Table 1 of GREN documentation was used to identify the new application classes to be created.

The next phase is to examine the hook methods to be overridden in the newly created classes. In order to better understand the concept of hook methods, consider Figure 3, which shows part of the GREN class hierarchy. All classes that contain methods in *italics* are abstract classes, and these methods need to be overridden in specialized classes. In fact, some of them are optionally overridden, depending on the framework usage. For example, the method *hasSourceParty* of the abstract class *BusinessResourceTransaction* needs to be overridden, while the method *sourcePartyClass* is optional, depending on whether the "Source Party" participant of the pattern Maintain the Resource (Figure 1) was used. This information can be obtained in the GREN documentation.

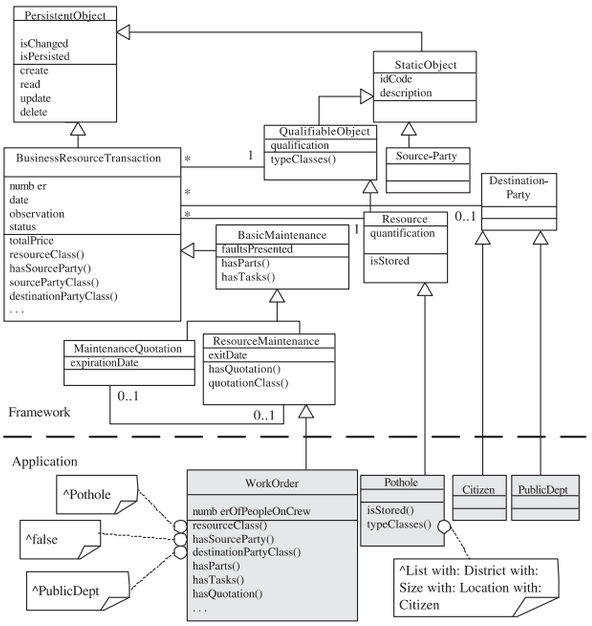


Fig. 3. Part of GREN class hierarchy and derived classes in the application

5 Implementation of the Specific Classes

The implementation of the specialized classes is done based on the list of classes and methods obtained in step 2 and using the same programming language used in the framework implementation. The first implementation activity is to create all the classes identified during the mapping, and their corresponding methods. Then, additional attributes and their methods are implemented. Finally the new application is compiled. According to the particular framework, it may be necessary to implement additional classes or methods, e.g. to deal with the graphical user interface.

In the GREN documentation there are several algorithms to help in the new classes creation, adaptation of the GUI to include new attributes, and production of the new system menus. In Figure 3 it can be observed some hook methods overridden in the PHTRS example. For example, the `Pothole` class inherits from *`Resource`*, which in turn inherits from *`QualifiableObject`*. So, according to Figure 3, the *`typeClasses`* method was overridden.

6 Test of the Resulting System

The resulting system needs to be tested to ensure both that the requirements have been fulfilled and that the system works in the end user environment. Based

on the compiled application code, on the framework documentation, and using an appropriate environment and test strategy, a tested application code to be deployed is produced.

In GREN, the testing step requires the installation of the framework software at a client machine, and the creation of a MySQL database to persist objects. The specific tables to be created are defined according to the patterns used during instantiation. The GREN cookbook has a special section to help its users to create such tables.

We have spent about 9 hours to obtain the final PHTRS system (2 hours in analysis, 1 hour in mapping, 4 hours in implementation, and 2 hours in testing). About 1600 lines of code (loc) were written to adapt GREN to this specific example (notice that about 50% of this code was automatically generated by VisualWorks GUI painters' tools and the remaining code is very simple and, most times, reused through copy/paste). These numbers are very low if we consider the number of Function Points (FP) [14] of the resulting system, which is about 370 (we have used an adjustment factor of 0.9 to calculate the function points). To obtain the same functionality by programming the applications from scratch would require at least five times more LOC than it actually did using GREN, considering that Smalltalk requires twenty-one LOC per function point [15].

7 Concluding Remarks

The approach here proposed intends to ease framework instantiation using pattern languages. Frameworks built using our approach have their architecture influenced by the pattern language, which eases the instantiation of new applications. With the special documentation provided by our approach, framework users basically need to know about the pattern language usage in order to instantiate the framework. No technical knowledge about the framework implementation details is needed. The novel aspect of our proposal is to use the pattern language to allow a smooth transition from analysis to implementation of specific applications.

We have used GREN to develop several systems, among which are a car repair management system, a sales system, and a video rental store. We have also conducted a case study with undergraduate students to implement a car rental and a hotel system. GREN attended the functionality of all these systems. With the case study presented in this paper it was also possible to confirm that GREN can be used to model aspects not envisioned before: by examining the syntax aspects of the patterns, we can achieve even more reuse than if we consider only the semantic aspects.

Although we have tested our approach only for small/medium applications, we believe that it scales well for larger applications. Due to the intrinsic characteristics of patterns, which allow the development of complex systems by partitioning them into smaller units, pattern languages can be extended by adding new patterns to cover more and more aspects of the domain. We are aware that GRN and GREN cover a restrict domain inside information systems, but

our approach can be adapted to allow its usage in a wider variety of domains. Extensibility for other computer science areas other than information systems could be an object of future research.

An important result of our approach to improve reuse is that the framework user knows exactly where to begin and to finish the instantiation, as the pattern language guides her/him through the several parts that need to be adapted in the framework. This solves a common problem of other approaches, because here the instantiation is focused on the functionality required, with a clear notion of which requirements are attended by each pattern.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994. 122

[2] D. Brugali and G. Menga. Frameworks and pattern languages: an intriguing relationship. *ACM Computing Surveys*, 32(1):2–7, March 1999. 122, 123

[3] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object Oriented Programming*, 1(2):22–35, jun/jul 1998. 122

[4] M. E. Fayad and R. E. Johnson. *Domain-Specific Application Frameworks: Frame-Works Experience by Industry*. John Wiley & Sons, New York, USA, 2000. 122

[5] A. Aarsten, D. Brugali, and G. Menga. *A CIM Framework and Pattern Language*, pages 21–42. Domain-Specific Application Frameworks: Frameworks Experience by Industry, M. Fayad, R. Johnson, –John Willey and Sons, 2000. 123

[6] R. E. Johnson. Documenting frameworks using patterns. In *OOPSLA '92*, pages 63–76, 1992. 123

[7] D. Gangopadhyay and S. Mitra. Understanding frameworks by exploration of exemplars. In *International Workshop on C. A. S.E*, pages 90–99, IEEE, July 1995. 123

[8] W. Pree, G. Pomberger, A. Schappert, and P. Sommerlad. Active guidance of framework development. *Software - Concepts and Tools*, 16(3):136–, 1995. 123

[9] A. Ortigosa and M. Campo. Towards agent-oriented assistance for framework instantiation. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2000. 123

[10] R. T. V. Braga and P. C. Masiero. A process for framework construction based on a pattern language, 2002. Proceedings of the 26th Annual International Computer Software and Applications Conference, IEEE computer Society, Oxford-England, to appear. 123, 128

[11] R. T. V. Braga, F. S. R. Germano, and P. C. Masiero. A pattern language for business resource management. In *6th Pattern Languages of Programs Conference (PLoP'99)*, Monticello – IL, USA, 1999. 123, 124

[12] R. T. V. Braga. GREN: A framework for business resource management. ICMC/USP – Sao Carlos, August 2001. Unpublished, Available on August, 2001 at: `http://www.icmc.sc.usp.br/~rtvb/GRENFramework.html`. 123

[13] R. S. Pressman. *Software Engineering - A Practitioner's Approach, 5th ed.* McGraw Hill, 2001. 126

[14] A. J. Albrecht. *AD/M Productivity Measurement and Estimate Validation*. IBM Corporate Information Systems, Purchase-NY, USA, 1984. 130

[15] C. Jones. *Preliminary Table of Languages and Levels*. Software Productivity Research Inc., Burlington, Mass., USA, 1989. 130

IS Components with Hyperclasses

Slim Turki and Michel Léonard

MATIS Geneva Team - Database Laboratory
Centre Universitaire d'Informatique, Université de Genève
24, rue du Général Dufour, CH-1211 Genève 4, Switzerland
{Slim.Turki,Michel.Leonard}@cui.unige.ch

Abstract. In this paper, we define an Information System component (ISC) in a data-intensive application context. An ISC is considered as an autonomous IS, respecting a set of completeness conditions. It is defined through a static space, a dynamic space and an integrity rules space. We use the hyperclass concept to implement the static space. A hyperclass is a large class, formed from a subset of conceptual classes of the global schema of a database, forming a unit with a precise semantics. This concept introduces a kind of modularity in the definition and the management of a database schema and a powerful kind of independence between the methods and the schema. Applied to a ISC, it facilitates its handling when refined or integrated.

1 Introduction

Several researches and industrial projects have addressed reuse in Information Systems (IS) Engineering, and several object-oriented approaches, such as patterns [19], frameworks [7], business components [7], artefacts, domain components [18], COTS, a-s-o. have been proposed to support reuse at different phases of software and IS development processes.

Nevertheless, most of the approaches about software components are intended to a process-intensive application context. In these components, only "public" functionalities are considered, whereas their internal parts are kept "private". The software component is often viewed as a black box with entries and results. Even if this model of component can be useful in some parts of an IS, it seems to us too restrictive. Indeed an IS component (ISC) have to share information stored in databases with other components. Its internal part concerning the manipulated information can't be "private" without hiding some important points in the IS design about precisely sharing information with other ISCs. Furthermore, assembling ISCs to build an IS is not only a question of plug-in: an ISC will use information that is available also in other ISCs and a necessary coordination between them must be established.

In another hand, the existing approaches address only one level of the IS (domain level, analysis and specification level or implementation level).

In our work, we are trying to create an environment more suited to the ISCs. Potentially it provides the basic mechanism to define, store, modify and integrate

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 132-141, 2002.

ISCs. An important point of this research is not to separate the different levels of an IS but on the contrary to study them in a whole approach.

In this paper, we present our approach for reuse in IS (Section 2) and give our definition of an ISC (Section 3). For us, an ISC is an implementation of a corresponding pattern and business object in an IS. It is built from a hyperclass (Section 4), its dynamic specifications and its integrity rules.

The concepts introduced in this paper are already or being implemented in our CAISE environment M7. M7 is based on a repository composed of 7 conceptual spaces (classes, hyperclasses, rules, life cycles, periods, events, treatments) that drives databases (DBs) running on a commercial DB management system (DBMS, Oracle8i).

2 Our Approach of Reuse

Our work is to be positioned in a data-intensive application context (in opposition with a process-intensive application context (names issued from [23]). In such data-centric approach, an IS component (ISC) interact with the other parts of the IS through a database (DB) to store, retrieve, manipulate and share data and information. Then, no part of the ISC should be kept hidden. In such context, the black box paradigm with the input/output model is too restrictive, even if this model of component can be useful in some parts of an IS.

In fact, in a software component (SC), only "public" functionalities are considered, whereas its internal parts are kept "private". The SC is often viewed as a black box with entries and results. Indeed an IS component have to share information stored in databases with other components. Its internal parts concerning the manipulated information can't be "private" without hiding some important points in the IS design about precisely sharing information with other ISCs. Furthermore, assembling ISCs to build an IS is not only a question of plug-in: an ISC will use information that is available also in other ISCs and a necessary coordination between them must be established.

In another hand, we support that to be effective, a component based development of an IS have to address in a global approach the different levels of the IS. This is why we are working on an evolutive IS framework, where we consider three levels:

1. Business objects (BO), at the domain level: set of concepts, rules, outstanding events… representing a sum of knowledge, know-how and skills in a knowledge base;
2. Patterns, at the analysis/specification level: A pattern is defined as a solution to a problem in a context [19]. It describes a generic situation in the IS development, or in front of which a responsible for the IS may be placed. It provides some generic possibilities with their advantages and drawbacks to take the good decisions and overcome the situation;
3. IS components (ISC), at the implementation level: an ISC is the implementation of a corresponding BO and pattern.

In this paper, we address only the third level: the ISC.

An ISC can be extracted from an existing DB schema (by reuse approach) or built from scratch (for reuse approach). It can be used as a base for a specification being made or as an element to be integrated in an existing or under construction DB or IS.

3 IS Component

Before giving a definition of an ISC, we return on the definition of an IS.

The object-oriented methods introduce the concept of objects gathering structure of data, controls and processing and propose total specifications of IS via static and dynamic complementary specifications [16]. For us, an IS is defined by a triplet {Static Space (*SS*), Dynamic Space (*DS*), Integrity Rules (*IR*)}.

IS=<SS, DS, IR>

3.1 Static Space

In this space are specified the static proprieties of an IS using a class diagram. In the class diagram are detailed the classes of the IS, their attributes, their keys, associations, and specialization/generalization relationships. Class methods are defined in this space, too.

Relationships between the classes are defined as referential dependencies, as existential dependencies, or as specialization dependencies. A specialization dependency is a special case of existential dependencies and an existential dependency is a special case of referential dependencies.

3.2 Dynamic Space

This model describes the specifications of the dynamic structures used in the life cycles and event structures. Dynamic specifications allow structuring an object-life (states) or an event structure (events and conditions).

There are several dynamic spaces in an IS specification such that the state-diagrams or the object life cycles described with state charts (UML) or Petri nets (M7).

Now we are implementing bipartite nets to specify the dynamic space of a component. This network contains nodes and stars, where nodes correspond to classes and stars to transactions.

A transaction is a complete cycle of data processing, which is carried out in response to an event or a change in the state of one or more objects of classes in entry of the transaction. A transaction constitutes a whole in itself, with a beginning and an end. It is a sequence of operations; treated as a single unit and that uses predefined primitives (elementary operations: Create, Retrieve, Update, Delete) or invocate class method.

A star (S_i) in the bipartite net has one or more nodes/classes (NI_j) in entry, and one or more nodes/classes in exit (NO_k). If completed, the transaction corresponding to S_i interacts with objects of the classes of NI_j and produces new objects in NO_k.

We define the domain of a transaction tr_i as the union of the set of classes in entry of tr_i and the set of classes in exit of tr_i:

$$\mathbf{D_{tri}=\{NI_j\}\cup\{NO_k\}}$$

3.3 Integrity Rules

Integrity rules (or constraints) (IR) are one of the cornerstones of an IS.

The role of IRs in a DB is to preserve its coherence, correctness and consistency during its exploitation. IRs must remain true for a database to preserve integrity. IRs are specified at DB schema creation time and enforced by the database management system.

- An IR is a condition defined on one or more classes, validated algorithmically.
- The context of an IR is the set of classes concerned by its definition or validation.
- The condition of one IR has to be checked for each state of the DB or change of state.
- The scope of an IR is the set of primitives of class modification that have to validate the IR to maintain the integrity of the DB.
- The response of an IR indicates the actions to be done in case of no-validation.

In addition to the referential and existential constraints defined in the static space, there are two types of IRs: static IR or behavioral IR. A static IR can be applied to each object of a class (static and individual IR) or to a set of objects of a class (static set IR). A behavioral IR is relative to an operation of modification of one or more objects of one class. It expresses a logical condition between the state of the DB before its modification and its state after its modification.

3.4 Definition of an IS Component

Here we define an ISC. For us, an ISC is a particular IS. So, as an IS, its definition is formed by a static space, where the data structure of the ISC is defined; a dynamic space, where the behaviour of the different elements of the ISC is expressed; and an integrity rules space, where rules governing the behaviour of the elements of the ISC are specified. In addition, the ISC has to be autonomous: it must contain all the elements that define it.

Formally, an ISC is defined as a triplet *{SSc, DSc, IRc}* respecting completeness conditions, necessary to guarantee its autonomy and coherence:

$$\mathbf{ISC=<SSc, DSc, IRc>}$$

1. *SSc* is a hyperclass (Section 4). This guarantees that the ISC has a semantic unity.
2. Class completeness: If a class Cl_i belongs to the ISC and references a class Cl_j, then Cl_j belongs to ISC. One reference between classes is a referential, existential or a specialization dependency.

$$\mathbf{\forall Cl_i \in SSc;\ Cl_i \rightarrow Cl_j \Rightarrow Cl_j \in SSc}$$

3. Transaction coherence: all the transactions of the ISC validate all the IRs of ISC.

4. Transaction completeness: if a transaction *tr* belongs to ISC, all the classes belonging to the domain D_{tr} of *tr* belong to ISC.

$$\textbf{if } tr \in DSc, \forall Cl_i \in D_{tr}; Cl_j \in SSc$$

5. Integrity rule completeness: all the classes belonging to the context C_{ir} of the IR *ir* belong to the ISC;

$$\textbf{if } ir \in IRc, \forall Cl_i \in C_{ir}; Cl_j \in SSc$$

3.5 ISC Handling

An ISC can be extracted from an existing DB schema or built from scratch. When defined, it can be used as a base for a specification being built or as an element to be integrated in an existing DB or IS.

To be able to adapt or refine one ISC or to integrate it, we should provide necessary operations to handle it. Classes, attributes and methods are grouped in one same category in the ISC. We choose/would like to select or unselect (classes/attributes/keys/methods/□). Furthermore, to create an environment to manage ISC, we are now implementing these operations for the handling of an ISC:

1. create an ISC,
2. extract an IS component from a database,
3. put an IS component into a database,
4. specialize/generalize an ISC,
5. modify an ISC,
6. condense an ISC,
7. expand an ISC,
8. study the common parts of two IS components.

To implement these ISCs handling operations, we are using the hyperclass concept to represent the static space of an ISC. In fact, when implemented, the concept of hyperclasses provide a DB Management System (DBMS) with a modularity of high level and improve its evolutionarily; it provides a powerful kind of independence between the methods and the schema.

In the next section, we present the concept of hyperclass and some related concepts such as hyperattributes, hyperobjects and hypermethods.

4 The Concept of Hyperclass

A hyperclass is a large class, formed from a subset of conceptual classes of the global schema of the DB, forming a unit with a precise semantics. Generally, it is associated with a function around the IS and it delimits the informational domain necessary for the achievement of this function. This space is a representation at the level of the DB schema of a field of particular competence around the IS.

Formally, a hyperclass HCl is defined by:

$$HCl=<SCl, RC, NGr>$$

1. $SCl=\{RC, Cl_1, Cl_2 \ldots Cl_n\}$. SCl is a subset of the set of the global schema classes;
2. *RC* is the root class of *HCl*. It is a class of *SCl* ($RC \in SCl$), and
3. *NGr* is the oriented navigation graph associated to *HCl*. This graph is composed of nodes corresponding one to one to the classes of *SCl*, and arcs corresponding to associations between these classes. It indicates how the objects of each class of *SCl* are reached from the object of *RC*. An arc from the class *A* to the class *B* in the informational schema indicates that one object o_A of *A* is linked to at most one object o_B of *B* and that inversely, o_B is linked to o_A. Furthermore, o_B may be linked to other objects of *A*. Thus, the navigation between the objects of related classes is independent from the direction of arcs in the informational schema.

A hyperclass is built in a similar way to one universal relation [22]; the links between two objects have the same semantics at the level of a hyperclass.

A hyperclass can share one or more of its elements with other hyperclasses. A HCl attribute is an attribute of a HCl class. The set of the attributes of HCl is the union of the sets of the attributes of the HCl classes.

For example, let's consider M@TIS (Fig.1.), an IS for a PhD program where teachings are generally given on different geographically distant sites.

To establish the end-of-year reports and the final student results, information about the students and the results of their courses are needed. This information is contained in the classes `STUDENT`, `PERSON`, `PROJECT`, `RESULT` and `COURSE`. These classes constitute the set of classes of the hyperclass `StudentResult` representing the informational domain needed to manage the academic results of the students. Student identifies the information about the students' results, and that's why `STUDENT` is the root of `StudentResult`.

Another important functionality in M@TIS is the calculation of the professors' salaries. Professors are paid according to the number of hours that they achieve and the university where they give each course. Hour-prices are not the same in all the universities of the training. Needed information for this functionality is available in the classes `PROFESSOR`, `PERSON`, `COURSE`, `PLANNING`, `CLASSROOM` and `UNIVERSITY`. Salaries will be established for each professor of the training. These classes will constitute the set SCl of classes of the hyperclass `PayProf` representing the informational domain needed to manage the professors' payments. Thus, the class `PROFESSOR` will be the root class (RC) of our hyperclass: its objects will identify the hyperobjects of `PayProf`.

4.1 Hyperobjects

The objects of a hyperclass HCl are called hyperobjects. A hyperobject ho is a set of linked objects, built starting from an object of the root class RC (its root object) and of the HCl closure of this root object.

The HCl closure of one hyperobject is the set composed of:

1. its root object o_{RC},
2. the objects of HCl classes linked to o_{RC}, and
3. the objects of HCl classes, which are linked to an object belonging to the closure.

The objects of the root class identify the hyperobject in a hyperclass.

The values taken by a hyperobject ho for an attribute *A* of a class *C* belonging to HCl are the union of the set of values taken for *A* by the *C* objects belonging to the HCl closure of *ho*.

The update of the value *v* taken by the HCl hyperobject *ho* for the attribute *A* of the HCl class *C* into the value *v'*, consists of updating the value *v* into *v'* taken for *A* by all the *C* objects of the HCl closure of *ho*.

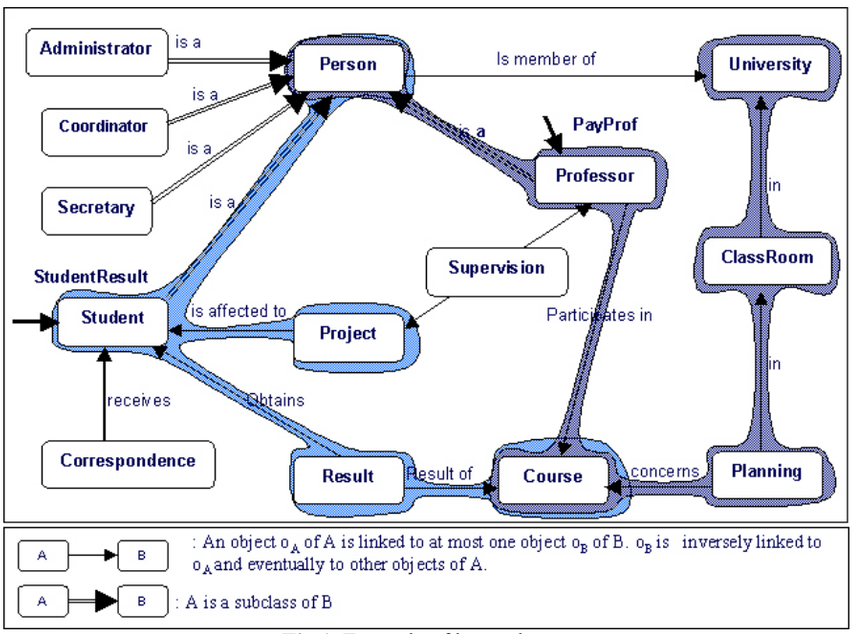


Fig.1. Example of hyperclasses

4.2 Hypermethods

Upon a hyperclass, it□s possible to define hypermethods, which use the hyperobject base methods (create, delete, retrieve, update).

A hypermethod Hm is defined over several classes of SCl. It can use different HCl attributes and invocate class methods of the HCl classes.

For example, on the hyperclass `PayProf`, we define a hypermethod called □`ProfSalary()`□ that calculates the professor□s salary from the number of hours that he taught in each university.

$$\text{PayProf}_i.profSalary() = \sum_{j,k[\text{planning}_j \,\&\, \text{university}_k \text{ linked in payProf}_i]} \text{planning}_j.duration * \text{university}_k.profHourCost$$

This hypermethod uses the attributes `PLANNING.duration` and `UNIVERSITY.profHourCost`. It manipulates the values taken for these attributes by the objects of the classes `PLANNING` and `UNIVERSITY` belonging to a same hyperobject.

4.3 Usefulness of the Concept of Hyperclass for ISCs

The hyperclass concept is useful in general when applied on a DB schema and particularly when handling ISC. In fact, this concept introduces a kind of modularity in the definition and the management of a DB schema. It is useful to facilitate the DB analysis, design, implementation and maintenance steps.

The traditional class concept of the object-oriented approach becomes a specialization of the hyperclass concept. It is a hyperclass with only a class, so the implementation of the hyperclass concept inside a DBMS does not require many new operations but only more sophisticate existing operations over classes (Create, Delete, Update, Retrieve), which were developed to support the class concept.

On another hand, hypermethods are an advantage for the method designers, because they don't have to choose a class for the method. Furthermore the hypermethod designers don't have to know the detailed schema of the hyperclass; they have to know only the attributes of the hyperclass: the mechanism of hyperclass undertakes to resolve access paths of each attribute of HCl each time that class attributes are accessed.

Furthermore, one hyperclass is a partial schema of the whole DB schema. Traditional evolution and modification methods are still available for this partial schema. Nevertheless, these methods must be adapted to the hyperclass. A set of schema evolution primitives is being developed. These primitives are supported by a mathematical theory [11, 12] and will permit a better handling of ISCs when to be refined or integrated.

As we showed it in [19], thanks to the hyperclasses, there is a powerful kind of independence between the data schema level and the method level. In fact, a priority of our work with hyperclasses is to preserve hypermethods from dysfunctions due to evolution operations: hypermethods must still functional (method conservation) and must still give the same result (method results conservation). A hypermethod Hm defined on a hyperclass HCl can be performed even after some HCl transformations, except of course, if they remove necessary information for Hm from HCl.

For example, if the class `CLASSROOM` is no longer useful in the hyperclass `PayProf`, we can hide it in `PayProf` and consolidate the hyperclass to maintain access to the class `UNIVERSITY`. Thanks to the hyperclass mechanism, ensuring access to hyperclass attributes, `ProfSalary` still functional and return the same results without any recompilation, even if `CLASSROOM` is excluded from `PayProf`.

Due to the independence between methods and schema, the hyperclass concept is very useful for the evolution of the static space of an ISC.

Generally, in an IS, dynamic space is implemented in a separate way from the static space. Previous work [11, 12] shows that there is a convergence between the implementation of the hyperclasses and the implementation of dynamic specifications. The Gavroche's clauses can be used to take the dynamic aspects into account in a hyperclass and the key point is the implementation of a bipartite net.

5 Conclusion

In this paper, we presented our global approach of reuse, built around three levels: business objects at the domain level; patterns at the analysis and specification level; and IS components, at the implementation level. In our research, these different levels are not separated but in contrary, they are studied in a whole approach.

An ISC is considered as an autonomous IS, respecting a set of completeness conditions. It is defined through a static space, a dynamic space and an integrity rules space. Hyperclasses are used to implement the static space.

Our work is to create an environment well suited to the ICSs. Potentially, and thanks to the hyperclass concept, it provides the basic mechanisms to modify ISCs and then to avoid to change every method of the IS component. In addition, this evolutive approach focuses on the information.

Targeted ISC will be generic and not dedicated to a particular technology, to a particular method, or to a particular IS.

References

1. Abrial J. R., Data semantics. J. W. Klimbie et K. L. Koffeman, IFIP TC2 Working Conference on Data Base Management, pages 1-59, Cargese (Corse), avril 1974. North Holland.
2. Backlund P., Patterns in Information System Engineering- An Initial Classification. In the proceedings of EMSAD□01, 6th CaiSE/IFIP8.1 International Workshop on Evaluation of Modelling Methods in Systems Analysis and Design, Interlaken, Switzerland, June 4-5, 2001.
3. Batini C., Lenrerini M., Navathe S. B., A comparative analysis of methodologies for database schema integration. ACM Computing Surveys, Vol. 18, No. 4, December 1986.
4. Bonjour M., Falquet G., Concept Bases: A Support to Information Systems Integration. In the Proceedings of CaiSE□94 Conference , Utrecht, 1994.
5. Ceri S., Fraternali P., Designing Database Applications With Objects and Rules : The Idea Methodology. Addison Wesley, Series on Database Systems and Applications. ISBN: 0201403692. October 1997.
6. Codd P., Object-oriented patterns. Communications of the ACM, P.152-159,Vol. 35, No. 9. September 1992.
7. Component-based enterprise frameworks. Communications of the ACM, P.25-66, Vol. 43, No. 10. October 2000.
8. Fankhauser P., Neuhod E. J., Incompleteness and explanation in Dynamic Schema integration. Position paper, Workshop on Heterogonous databases & semantic interoperability, 1992.
9. L□onard M., Parchet O., Information overlap. In the Proceedings of the 1999 International Symposium on Database Applications in Non-Traditional Environments, DANTE□99.

10. Léonard M., An evolutive approach for the IS design: the Gavroche model, Conf. Inforsid, La Garde (F), May 1999.
11. Léonard M., Pham Thi T. T., Conceptual model: an integration of static aspects and dynamic aspects of information system, Conference on IT 2000, Ho Chi Minh City, Vietnam.
12. Léonard M., Pham Thi T. T., Information System integration with the static and dynamic aspects, Swiss-Japanese seminar, Kyoto, December 1999.
13. Maier D., Ullman J. D., Vardi M. Y., On the foundations of the Universal Relation Model. ACM Transactions on Database Systems, Vol. 9, No. 2, June 1984, Pages 283-308.
14. Mirabel I., Un mécanisme d'intégration de schémas de conception orientée objet. Thèse de doctorat de l'Université de Nice-Sophia Antipolis, Décembre 1996.
15. Nierstrasz O. Gibbs S., Tsichritzis D., Component-oriented software development.Communications of the ACM, P. 160165, Vol. 35, No. 9. September 1992.
16. Oussalah Ch., Génie objet, analyse et conception de l'évolution. Hermes science publications, Paris 1999, ISBN 2-7462-0029-5.
17. Parent C., Spaccapietra S., Database Integration: an Overview of Issues and Approaches. Communications of the ACM, vol. 41, no 5, pp. 166-178, May 1998.
18. Semmak F., Réutilisation des composants de domaine dans la conception des systèmes d'information. PhD Thesis in Computer science, Université Paris I, Février 1998.
19. Software Patterns, Communications of the ACM, Vol. 39, No. 10, 1996.
20. Turki, S., Léonard, M., Hyperclasses: towards a new kind of independence of the methods from the schema. In the proceedings of the 4th International Conference on Entreprise Information Systems, ICEIS'2002, Vol.2, P. 788-794, ISBN: 972-98050-6-7. Ciudad-Real, Spain, April 3-6, 2002.
21. Turki, S., Introduction aux hyperclasses. In the proceedings of Inforsid'2001, P. 281-299, ISBN: 2-906855-17-0. Martigny, Suisse, Mai 2001.
22. Maier D., Ullman J.D., Vardi M. Y., On the foundations of the Universal Relation Model. ACM Transactions on Database Systems, Vol. 9, No. 2, June 1984, Pages 283-308.
23. Van den Berghe T., A Methodological Framework for Active-Application Development. PhD thesis, Institut d'Administration et de Gestion, Université catholique de Louvain, Belgium, October 1999.

Collaborative Simulation by Reuse of COTS Simulators with a Reflexive XML Middleware[1]

Mathieu Blanc[2], Fabien Costantini[1], Sébastien Dubois[1], Manuel Forget[1], Olivier Francillon[2], and Christian Toinard[1,3]

[1] CNAM-CEDRIC 292 rue Saint-Martin 75141 Paris Cedex 03, France
{costanti,toinard}@cnam.fr,
{dubois_se,forget}@iie.cnam.fr
http://cedric.cnam.fr/~toinard
[2] ENSEIRB BP 99 33402 Talence Cedex, France
{blanc, francill}@enseirb.fr
[3] ENSI Bourges 10 Boulevard Lahitolle 18020 Bourges Cedex, France

Abstract. Reuse of Commercial Off-The-Shelf simulators is a promising and new approach to support collaborative and distributed simulations. This paper presents ComRTI, a reflexive XML middleware that eases reuse and collaboration between various COTS simulators. It is a portable middleware that runs on Windows and Unix stations. ComRTI uses only widely available software such as TCP/IP and XML toolkits. Heterogeneous simulators cooperate by sharing a neutral XML model. ComRTI eases to manage meta-models and neutral models. It helps to project dynamically a neutral model onto a specific target simulator. ComRTI has proven efficiency to develop cooperation services between industrial designers. Thus, distributed designers cooperate easily by sharing meta-models and reusing their classical simulation interface. The designers can cooperate also by running a distributed simulation involving different meta-models.

1 Introduction

ComRTI is a reflexive XML middleware that eases integration and reuse between various COTS 3D modelers and simulators. CoSimul is a cooperative application that uses ComRTI to provide cooperative simulation services. ComRTI and CoSimul have been developed within the IST-AIT VEPOP project of the European Commission.

The ComRTI and CoSimul software answer several industrial requirements 1) easy coupling of 3D models with simulation models 2) reuse of COTS simulators and Computer Aided Design tools 4) use of XML 5) propose an extensible and reusable approach 5) use of off-the-shelf and widely available communication software.

[1] Work supported by the European Commission under contract Number IST 1999-13346, EADS Airbus, EADS Astrium, EADS CCR, CNAM-CEDRIC, Flow-Master, University of Oulu, DFKI.

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 142-151, 2002.

ComRTI and CoSimul address the lack of solution to couple easily COTS simulators (e.g. FlowMaster2 [3]) and 3D space allocation application (e.g. CATIA [5] or Rapid Prototyping software). Classical CAD system like CATIA enables to integrate simulators within the 3D scene. But, they do not provide any middleware to reuse external commercial simulators like FlowMaster2. Currently commercial CAD systems do not provide any service to set-up in a cooperative way a distributed simulation. Distributed systems like [6] aim at providing communication services to achieve a distributed simulation. But these systems do not ease the reuse of COTS simulators. Other solutions like [10] requires access to the numerical code of a simulator to be able to achieve a parallel simulation. Moreover, middleware like [8] and [13] provide communication architecture to develop client-server application. But, reflexive aspects are not integrated within the communication architecture. Management of meta-models and models is complex: it requires a projection of meta-modeling facilities onto useless intermediate language (e.g. CORBA-IDL). So, reflexive aspects are developed on top of the communication system with extra complexity.

In contrast, our solution does not require any access to the internal structure and computation of the simulator. It aims at reusing easily COTS simulators and 3D software. Reflexive aspects are integrated within our core communication services. Thus, extensible meta-modeling and dynamic sharing of models are provided with few complexity.

Let us summarize the benefits provided by a distributed application like CoSimul developed using ComRTI.

First, designers can cooperate by reusing COTS software. For example, they can share a neutral 3D model between heterogeneous 3D space allocation applications. Thus, they can cooperate and interact through heterogeneous CAD systems.

Second, designers can cooperate to set-up a distributed simulation by coupling easily 3D space allocation systems and COTS simulators. Thus, a simulation expert reuses easily existing 3D models. The simulation model is deduced automatically from the 3D model. The time to set-up a simulation is reduced and different designers, with complementary skills, cooperate by reusing easily different consistent simulation models.

Third, a distant designer reuses his simulator interface (e.g. the FM2 interface). He does not have to learn a new simulator interface to process various simulations starting from the received neutral model.

For the development of distributed simulation software, ComRTI provides different advantages.

First, it provides an open architecture to reuse COTS simulators. Using ComRTI, an adapter projects a neutral simulation model onto the target simulation model. A ComRTI adapter is a reusable software component. Thus, dynamic reuse and assembly of existing Adapters enables the coupling between simulators for various domains of simulation. Reuse of Adapters and COTS simulators is provided through standard and extensible methods to dynamically process various simulation meta-models and neutral models.

Second, ComRTI is independent of the target programming language and simulator. Thus the same neutral model can be shared between, for example, an adapter written in Java and a distant adapter using C++.

Third, distributed adapters share easily a neutral model using the ComRTI publication/subscription framework. That framework enables a dynamic propagation of a model instance beyond the interested adapters. Thus, peer-to-peer communications are easily developed using ComRTI publication and subscription methods.

Finally, mediators can be easily developed to achieve dynamic bi-directional transformations between various neutral models. For example, a mediator transforms automatically a neutral 3D model into a neutral flow simulation model. Thus, different business logics can reuse existing neutral models to carry out various bi-directional transformations.

Section 2 presents the Object Management Architecture proposed by ComRTI and the related components (i.e. Object Simulation Bus, Adapters, Simulators and Mediators). Section 3 presents the ComRTI Object Simulation Broker that is a reflexive kernel for developing communications between Adapters. Section 4 compares ComRTI with related works coming from CORBA, distributed simulation run-time and parallel computing. Conclusion describes the industrial feedback and future works.

2 Object Management Architecture of ComRTI

An Object Management Architecture defines the cooperation between the different software components involved within ComRTI. This solution answers two main objectives defined by our industrial partners.

- the possibility of making a generic parameter setting that is independent of the configuration of a simulator (e.g. FlowMaster2). This possibility is supported by the proposed solution that permits a distinction between a neutral model and attributes related to a target simulator.
- the possibility to dynamically chain simulators. It is possible using ComRTI through an adapted mechanism for propagating produced values to the consuming simulators. Various chaining, such as lightweight coupling (i.e. with loops) are supported. Consistent coupling (i.e. sophisticated synchronization between the simulators) can also be integrated. The important point is the ability to chain dynamically COTS simulators through reusable Adapters.

The ComRTI OMA defines a layered and well-decoupled architecture where 1) an Adapter achieves the projection of a neutral simulation model onto a target simulator model and 2) a Mediator (i.e. a specific Adapter with various business logics) translates a neutral model for a given Domain (e.g. 3D Geometry) to another neutral model for another Domain (e.g. Hydraulic). An Adapter uses the ComRTI Object Simulation Broker to plug dynamically a target simulator on the ComRTI simulation bus. Thus, an Adapter can connect easily with other Adapters to share an instance of a neutral simulation model.

The Object Simulation Broker includes 1) a Generic Adapter Layer that permits concrete adapters to use a meta-meta model and connection facilities (i.e. publication/subscription services) and 2) a Proxy Adapter Layer that enables to share any

instance of a neutral model (i.e. deriving from the meta-meta model) based on XML transmissions.

The Generic Adapter Layer in its turn relies on the ComRTI baseline communication and configuration services. This Generic Adapter Layer provides means to dynamically link Adapters through meta-modeling management.

The Proxy Adapter Layer enables to transmit any kind of model as an XML stream.

For interfacing a new COTS simulator with others, an implementer only needs to create a new Adapter. To interface with other domains of simulation, implementers need to further as well a Mediator between the existing domain and the new one. All the ComRTI parts are reusable components for dynamic meta-modeling and sharing. An Adapter is also a reusable component that can share a model instance with future Adapters.

ComRTI OSB is a simulation bus where the Adapters plug dynamically different neutral model instances. An application peer that subscribes to a type of meta-model (e.g. a Flow model) receives automatically a produced model instance of that type.

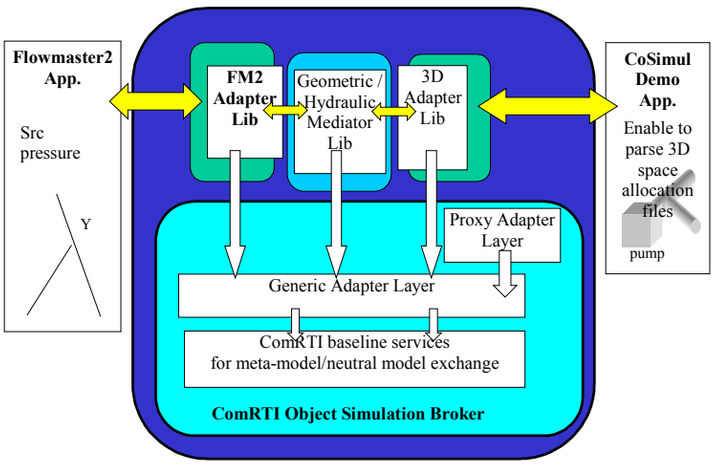


Fig. 1. ComRTI architecture used for the CoSimul application

3 ComRTI Object Simulation Broker

3.1 Generic Adapter Layer

The general problem of integration of COST simulators is that each simulator has a proprietary model for describing simulation networks.

In order to integrate two simulators A and B, the first solution is to make direct conversions between the two proprietary models A and B. This solution is not satisfying since the number of converter is N^2.

The second solution is to achieve the conversion using a pivot model (i.e. a *Neutral model X*) that is independent of *Model A* and *Model B*. It is a better solution: once the Adapter, to achieve the conversion between the Model A (i.e. a target simulator) and

the Neutral model X is available, the connection with any other proprietary model is possible. The number of Adapters is N. So, there is much less Adapters to develop. Moreover, an Adapter is independent of the other Adapters since it uses a pivot model (i.e. a neutral model) and projects that neutral model onto the target simulator model. ComRTI uses that approach through a reflexive kernel (i.e. the Generic Adapter Layer) that enables to define all the required meta-model and neutral models.

As depicted in Fig. 2, a Concrete Simulator Adapter inherits from the Generic Adapter Layer class. It refers to the publication and subscription services of the Generic Adapter Layer in order to dynamically link, synchronize with other Concrete Simulator Adapters. The Generic Adapter Layer is a reflexive component that permits to describe any type of meta-model. So, the Generic Adapter Layer provides a meta-meta model that describes itself and enables to define all the required meta-models and shareable instances of neutral model. Thus, the number of Adapters is limited to the number of target simulators due to a pivot neutral model.

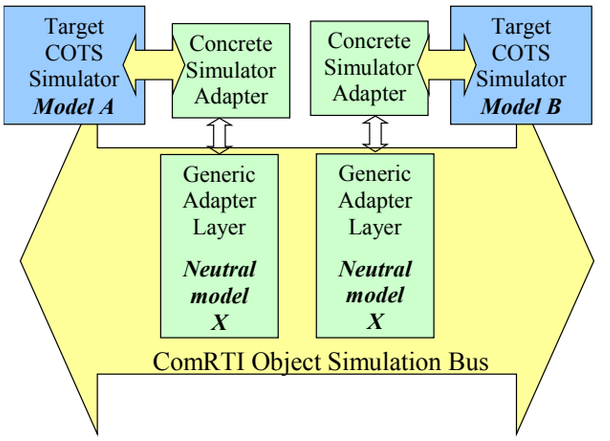


Fig. 2. Concrete Simulator Adapter inherits from the Generic Adapter Layer to project the shared neutral model onto a target simulator model

The Producer/Consumer Scheme Used by the Adapters. In order to define how the simulators are interconnected and what they exchange between them, the architecture furthers a dynamic approach for simulators interconnection. It is the role of the Generic Adapter Layer to connect the adapters depending on their domains of interest, whether they are producers of data models, consumers of data models or both, whether they can produce simulation and/or results. It does this dynamically. Typically, the first thing a concrete adapter does is to register to its Generic Adapter Layer. The registration process permits to dynamically link the concrete adapter to the others via the layered architecture.

The Proxy Adapter Protocol. Fig. 3 describes the transmission protocol between a producing peer and a distant consuming peer. First, the Concrete Adapter on peer A publishes a DataModel. Second, the GAL notifies the Proxy Adapter that a DataModel is available. Third, the Proxy Adapter converts the consumed DataModel into

an XML stream and sends it to the distant Proxy Adapter. Fourth, the distant Proxy Adapter publishes the received DataModel. Finally, the GAL notifies the distant consuming Concrete Adapter.

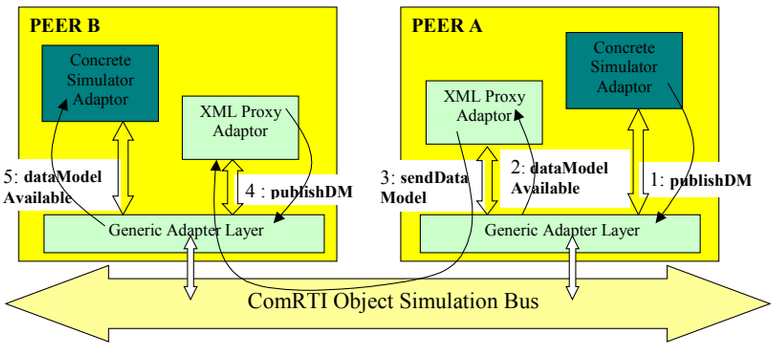


Fig. 3. Interaction protocol between distributed peers

The Proxy Adapter Layer implements the conversion of a model instance into an XML stream. The Proxy Adapter Layer is fully generic. It is able to convert any type of model into an XML stream. The Proxy Adapter Layer achieves TCP/IP transmissions and thread management using the portable ACE toolkit [1].

3.2 Reflexive Principles

The Meta-meta Level. The Generic Adapter Layer includes the meta-meta level of our reflexive middleware. The meta-meta level provides reusable *metaclasses* that support a tree representation of any meta-model/neutral model. Thus, the components of a model and the attributes of each component have a standard representation parseable by all adapters.

The Generic Adapter Layer provides several metaclasses.
Every meta-model inherits from the `DataModel` metaclass whose attributes are:

- `type` : the type (most of the time related to the domain) of the meta-model,
- `cdata`: the data representation whose type is `CompDataObj`.

Every component of a meta-model inherits from the `CompDataObj` metaclass. That metaclass is based upon the Composite Design Pattern [4]. The Composite Pattern `CompDataObj` enables to implement `DataObj` trees in the Data Models (Fig. 5).

Each `DataObj` can contain attributes. The attributes are also implemented with the Composite Design Pattern [4]. Fig. 4 illustrates that composite classes hierarchy.

- `AttrValue` is the leaf of the Composite Attribute Pattern. It enables to manipulate dynamically a scalar attribute (i.e. dynamic typed attribute creation).
- `Attribute` inherits from AttrValue and adds a name.
- `CompAttr` is the composite (recursive) part. It permits to represent non-scalar and hierarchically organized attributes.

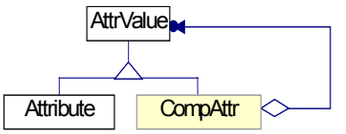


Fig. 4. The Composite Attribute Design Pattern implementation class hierarchy

The Meta Level and the Model Level. Fig. 5 presents the application of the `CompDataObj` metaclass to define an hydraulic meta-model called `HydroComp` that is a real composite for the domain of Hydraulic Simulation.

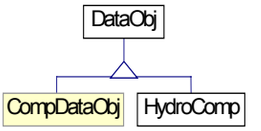


Fig. 5. Application of a concrete adapter defining an Hydraulic meta-model

Fig. 6 shows that `HydroComp` includes real objects such as `SrcPressure` and `Pipe` deriving from `DataObj`.

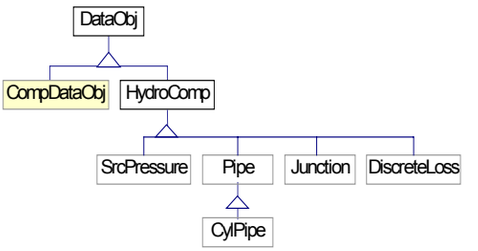


Fig. 6. Application to define a neutral model at a concrete hydraulic simulator side

Fig. 7 presents the XML tree resulting from the serialization of an instance of the previous hydraulic neutral model. The ProxyAdapter carries out that serialization. One can see that a `DataModel` includes a `CompDataObj` that defines a hierarchy of hydraulic `DataObj`. Each hydraulic `DataObj` includes a category (i.e. "HydroNet") from the meta-model and a name (e.g. "pipe:cylindrical") from the neutral model.

4 Related Works

MOF [9] can be used in conjunction with the CORBA-ORB [8]. A MOF model is thus projected onto a CORBA-IDL interface. But it requires to compile the IDL interface in order to instanciate the model. The IDL compilation provides C++ or Java skeletons for developing remote method calls. So, meta-modeling is supported on top of a CORBA bus. But the IDL projection does not simplify the development process. Moreover, CORBA interactions are client-server based and sharing is poorly sup-

ported by additional CORBA Services such as the Event Service. In contrast, ComRTI manages directly meta-modeling through its reflexive kernel. So, intermediate projection (e.g. IDL projection) is useless. Moreover, XML transmission solves the heterogeneity of the machines. Model sharing is supported using a peer-to-peer approach based on efficient subscription and publication services for transmitting an XML tree.

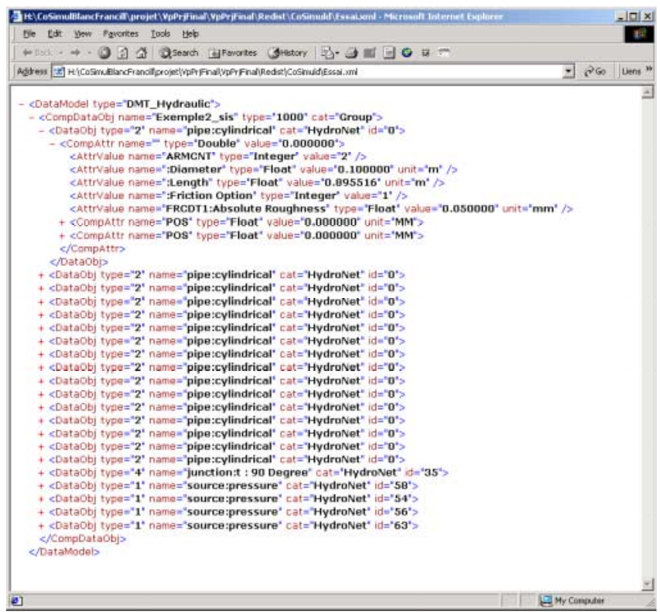


Fig. 7. The XML tree corresponding to the serialization of a DataModel tree

High Level Architecture [6] is a distributed system devoted to simulation that is also adopted as a CORBA standard [7]. It comes from military requirements to define distributed simulation of battlefields. But, HLA provides a low-level communication approach that does not help to integrate COTS simulators. The HLA-OMT language is devoted to the standardization of distributed interactions but it does not support meta-modeling. Moreover, HLA aims at guarantying reproducibility and consistency of the distributed simulation using a logical time ordering. But, the only tentative [11] with COTS simulators show that HLA fails to satisfy those properties. In practice, HLA requires access to the internal computation of a simulator. In contrast, ComRTI eases the reuse and the assembly of COTS simulators through meta-modeling facilities. It provides a reusable and extensible reflexive architecture as inner meta-modeling facilities.

Jaco3 [10] addresses clearly the parallel computation of a global numerical code. The main code uses distant CORBA objects and acts as a CORBA client that calls an object to compute in parallel operations on matrixes (i.e. a multiplication or an addition). So, the main task calls a parallel CORBA object. That project claims also to address reuse of existing simulation software. But the proposed demonstrator is a parallel execution of an existing simulator. So, despite they claim to address reusability the proposed demonstrator is not relevant of this point. ComRTI provides a com-

plementary approach. It addresses clearly reuse and integration of existing simulators. The CoSimul demonstrator shows the efficiency of the ComRTI approach to reuse COTS simulators through reflexive meta-modeling and XML streams.

5 Conclusion

ComRTI provides a reflexive Object Simulation Broker that is devoted to the dynamic coupling and reuse of COTS simulators. It provides meta-modeling facilities. It supports cooperation between heterogeneous software through the sharing of neutral simulation model instances based on XML transmissions.

ComRTI has been used successfully to develop the CoSimul application. Thus, designers can cooperate by sharing neutral simulation models starting from their usual application interface to reuse existing CAD models. Thus, they can easily set-up a distributed simulation involving multiple sites and simulation networks (e.g. flow computation for a ventilation network and fuel network) by reusing 3D models. Distributed results can be shared to carry out simulation loops or cooperative improvements.

Industrial feedback shows the powerful of CoSimul/ComRTI to reuse and assemble COTS simulators. The time to set-up a simulation is reduced and various cooperation scenarios are supported. The reuse capabilities of CoSimul/ComRTI provide major benefits 1) Designer can reuse their usual application interface 2) Cooperation is supported through heterogeneous applications 3) Designer reuse CAD models to set-up quickly a simulation 4) ComRTI provides reusable and extensible classes 5) Adapters are reusable components and 6) Adapters process and share reusable neutral models.

Future works will consist in communication improvements. First, new Mediators will be developed to achieve automatic and lightweight coupling between different simulation domains. Second, CTOP [14] communications will be integrated to guaranty consistency of a distributed simulation. Third, DBSM services [3] will enable the cooperative modification of a model instance based upon mobile services for managing a distributed model tree. Thus, the solution will support better the mobile working that an extended cooperative team requires.

References

1. Box, D. F. , Schmidt, D. C., Suda, T.: ADAPTIVE: An Object-Oriented Framework for Flexible and Adaptive Communication Protocols, Proceedings of the 4th Conference on High Performance Networking, IFIP, Liege, Belgium, December 14-18 (1992) 367-382
2. Costantini, F. , Toinard, C.: A New Approach of Collaborative Learning with the Distributed Building Site Metaphor. IEEE MultiMedia, Vol. 8, No3, July-September (2001) 21-29
3. FlowMaster International: FlowMaster 2 Reference Help 6.2. http://www.flowmaster.com (2002)

4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns Elements of Reusable Object-Oriented Software. Addison Wesley (1995)
5. IBM: CATIA V5 Latest Release, http://www-3.ibm.com/, (2001)
6. IEEE Computer Society: IEEE Std 1516-2000, IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules. (2000)
7. Object Management Group: CORBAmanufacturing: Manufacturing Domain Specifications. Version 1.0 (1999)
8. Object Management Group: The Common Object Request Broker - Architecture and Specification - Edition 2.3 (1999)
9. Object Management Group: Meta Object Facility (MOF) Specification (Version 1.3 RTF) http://www.omg.org/cgi-bin/doc?ad/99-09-04.pdf (1999)
10. Perez, C., Priol, T.: Grid Computing with Off-the-Shelves Middleware Technologies. ERCIM News No.45 - April (2001)
11. Schulze, T., Straßburger, S., Klein, U: Migration of HLA into Civil Domains: Solutions and Prototypes for Transportation Applications. SIMULATION, Vol. 73, No. 5, pp 296-303, November (1999)
12. Toinard, C.: CTOP: un service de diffusion ordonnée causalement et totalement fonctionnant sur Internet. Actes Nouvelles Technologies de la Répartition, NOTERE 2000, 22-24 Novembre, Paris, Disponible aussi dans la revue électronique RERIR (2000)
13. W3C: Simple Object Access Protocol (SOAP) 1.1, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/ (2000)

Efficient Web-Based Information Systems

Omar Boucelma[1] and Zoé Lacroix[2]

[1] Université de Provence - LSIS, France
[2] Arizona State University, USA

Preface

Many approaches have been developed in the past to address efficient query processing for information systems, integrated systems, as well as Web-based systems (caching, etc.). However new techniques need to be developed to optimize query processing for Web-based information systems (WIS). These new techniques need to address several challenges including management and publication of semi-structured data, limited access to Web data sources, lack of published information about their content (statistics, metadata), complex coverage, etc.

This is the first of (hopefully) a successful series of workshops dedicated to efficiency measurements and enforcement of Web-based information systems. The EWIS workshop aims to bring together researchers and practitioners to discuss the use and the development of next generation WIS, ranking from simple Internet/Intranet-based ones to much more dedicated ones such as GRID platforms, which allows to couple geographically distributed resources with an inexpensive access to these resources irrespective of their physical location or access point.

The program consists mainly of a scientific track, of which the papers that appear in these proceedings. For this track the program committee selected 6 papers out from the fifteen submitted. In addition, the workshop program includes an invited paper that describes the ARTEMIS/MOMIS system for semantic integration and query optimization of heterogeneous data sources.

Organization

This workshop is organized by Omar Boucelma (Université de Provence, France) and Zoé Lacroix (Arizona state University, USA).

Program Committee

M. Adiba (IMAG, Fr)
S. Bressan (NUS, Sg)
G. Dobbie (U. Auckland, NZ)
W. Fan (U. Temple, USA)
J. Freire (Bell Labs, USA)
I. Manolescu (INRIA, Fr)
L. Raschid (U. Maryland, USA)
M. Rys (Microsoft, USA)

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 152–153, 2002.

M. Scholl (CNAM, Fr)
D. Srivastava (ATT Res. Labs, USA)
K. Stoffel (U. Neuchâtel, CH)
L. Tanca (Politecnico di Milano, It)
Z. Tari (RMIT, Au)

Additional Referees

L. Bright (U. Maryland, USA)
U. Nambiar (ASU, USA)
T. Lim (U. Auckland, NZ)

Sponsoring Institutions

LSIS, Marseille

Primary Contact

For more details on the workshop please contact:

Omar Boucelma
Université de Provence - LSIS
39, rue F. Joliot-Curie
F-13453 Marseille Cedex 13
omar@cmi.univ-mrs.fr

Zoé Lacroix
Arizona State University
PO Box 87610
Tempe AZ 85287-6106, USA
zoe.lacroix@asu.edu

Semantic Integration and Query Optimization of Heterogeneous Data Sources*
(Invited Paper)

Domenico Beneventano[1], Sonia Bergamaschi[1], Silvana Castano[2], Valeria De Antonellis[3], Alfio Ferrara[2], Francesco Guerra[1], Federica Mandreoli[1], Giorgio Carlo Ornetti[2], and Maurizio Vincini[1]

[1] Università di Modena e Reggio Emilia
{beneventano.domenico,bergamaschi.sonia,guerra.francesco, mandreoli.federica, vincini.maurizio}@unimo.it
[2] Università di Milano
{castano,ferrara,ornetti}@dsi.unimi.it
[3] Università di Brescia
{deantone}@ing.unibs.it

Abstract. In modern Internet/Intranet-based architectures, an increasing number of applications requires an integrated and uniform access to a multitude of heterogeneous and distributed data sources. In this paper, we describe the ARTEMIS/MOMIS system for the semantic integration and query optimization of heterogeneous structured and semistructured data sources.

1 Introduction

In modern Internet/Intranet-based architectures, an increasing number of applications requires an integrated and uniform access to a multitude of heterogeneous and distributed data sources. A methodological framework for the integration of highly heterogeneous data sources requires guidance and support tools for dealing with different types of heterogeneity that can occur among the sources, and should provide intelligent techniques and tools for automating as much as possible various activities involved in the semantic integration process, such as for example the acquisition of interschema properties, the schema matching and reconciliation activities, or the generation of semantic mappings between the global schema and the local schemas of the underlying data sources for query purposes. A first type of heterogeneity to be considered when integrating heterogeneous sources derives from the level of structuring of data to be integrated. Several data models and languages can be adopted for data representation and storage, ranging from non structured data, typical of file systems, to highly structured data, typical of database systems, to semistructured data, typical of Web data sources [8]. Another type of heterogeneity to be considered derives from the

* This paper has been partially funded by MIUR COFIN2000 D2I project.

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 154–165, 2002.

terminology, structure, and context characterizing data stored at each source, which requires to take into account data semantics and to define semantic mappings between data. These problems have been first addressed in the context of database integration [9], and more recently in the context of Web and XML data integration [10, 5, 6].

In this paper, we describe the ARTEMIS/MOMIS system for the semantic integration and query optimization of heterogeneous structured and semistructured data sources [2, 3]. ARTEMIS/MOMIS supports a virtual approach to integration, in that data residing at the sources are accessed during query processing by exploiting defined mappings, instead of being replicated at the global level. The global schema, associated mapping descriptions and interschema knowledge obtained from the integration process provide support for expressing queries in terms of the global schema over underlying data sources, with mechanisms for the reformulation and answering of such queries in terms of the data stored in the sources. Such a problem is known in the literature as view-based query processing, and has been studied very actively in the recent years[7, 11, 12].

The paper is organized as follows. Section 2 describes the interschema knowledge extraction and representation process. Section 3 is devoted to the source schema matching process. Section 4 describes the global schema and mapping generation process. Section 5 illustrates the semantic query otimization process. Finally, in Section 6, we give our concluding remarks.

2 Interschema Knowledge Extraction and Representation

The semantic integration of strongly heterogeneous data sources is performed by constructing a semantically rich representation of the data sources to be integrated, by means of a common data model based on the ODL$_{I^3}$ language. In ARTEMIS/MOMIS, this task is performed by wrapper tools developed for both structured and semistructured data sources [2]. The subsequent phase of the integration process is the extraction and representation of the interschema properties featuring the sources to be integrated. In ARTEMIS/MOMIS, interschema knowledge is expressed through *intensional properties* and *extensional properties*. Intensional properties are *terminological relationships* expressing inter-schema knowledge at the intensional level for the source schemas. Intensional relationships are defined between classes and attributes, and are specified by considering class/attribute names, called terms. The following relationships can be specified in ODL$_{I^3}$: SYN (Synonym-of), BT/NT (Broader/Narrower Terms), or hypernymy/hyponymy, and RT (Related Terms), or positive association [2, 3].

An intensional relationships is only a terminological relationship, with no implications on the extension/compatibility of the structure (domain) of the two involved classes (attributes). For capturing this kind of knowledge, extensional properties are used. Extensional properties express interschema knowledge at the extensional level. The intensional relationships SYN, BT, NT and RT between two classes C_1 and C_2 may be "strengthened" by establishing that they are also *ex-*

UNIVERSITY source (UNI)

```
Research_Staff(name,e_mail,dept_code,s_code)
School_Member(name,school,year,e_mail)
Department(dept_name,dept_code,budget)
Section(section_name,s_code,length,room_code)
Room(room_code,seats_number,notes)
```

COMPUTER_SCIENCE source (CS)

```
CS_Person(first_name,last_name)
Professor:CS_Person(belongs_to:Division,rank)
Student:CS_Person(year,takes:set⟨Course⟩,rank,e_mail)
Division(description,address:Location)
Location(city,street,number,country)
Course(course_name,taught_by:Professor)
```

TAX_POSITION_XML source (TP)

```
<!ELEMENT ListOfStudent (Student*)>
<!ELEMENT Student (name, s_code, school_name, e_mail, tax_fee)>
<!ELEMENT name (#PCDATA)>
...
```

Fig. 1. Three heterogeneous University sources

tensional relationships [4]. Consequently, the following extensional relationships can be defined in ODL_{I^3}:

- C_1 SYN_{ext} C_2: the instances of C_1 and C_2 are the same.
- C_1 BT_{ext} C_2: the instances of C_1 are a superset of the instances of C_2.
- C_1 NT_{ext} C_2: the instances of C_1 are a subset of the instances of C_2.
- C_1 RT_{ext} C_2: the instances of C_1 overlap the instances of C_2.
- C_1 DISJ_{ext} C_2: the instances of C_1 are disjoint from the instances of C_2.

Intensional and extensional properties can be partially automatically extracted and partially explicitly declared by the integration designer. A *Common Thesaurus* of terminological and extensional relationships is constructed, describing interschema knowledge about ODL_{I^3} classes and attributes of source schemas.

The Common Thesaurus is built through an incremental process during which relationships are added in the following order: *i) schema-derived relationships*: intensional and extensional relationships holding at intraschema level are extracted by analyzing each ODL_{I^3} schema separately; *ii) lexical-derived relationships*: intensional relationships holding at interschema level are extracted by analyzing different sources ODL_{I^3} schemas together according to the Wordnet supplied ontology; *iii) designer-supplied relationships*: additional intensional and extensional relationships are supplied directly by the designer, to capture domain knowledge about the source schemas. Supplied relationships are validated with ODB-Tools [1]; *iv) inferred relationships*: a new set of relationships

(i.e. new generalization and aggregation properties) is inferred by ODB-Tools by reasoning over the union of the local schemas enriched with currently available interscheme properties. As a running example, we consider three heterogeneous sources (see Figure 1): the first source `University` (UNI) is a relational database; the second source `Computer_Science` (CS) is an object–oriented database; the third source, `Tax_Position` (TP), is an XML file. Possible extensional relationships for the sources of the running example are the following:

1. `UNI.School_Member` SYN_{Ext} `TP.Student`
2. `CS.Student` NT_{Ext}`UNI.School_Member`
3. `CS.Professor` NT_{Ext} `UNI.Research_Staff`
4. `CS.Professor` DISJ_{Ext} `UNI.School_Member`
5. `UNI.Research_Staff` DISJ_{Ext} `TP.Student`
6. `UNI.Research_Staff` DISJ_{Ext} `CS.Student`
7. `CS.Student` NT_{Ext} `CS.CS_Person`
8. `CS.Professor` NT_{Ext} `CS.CS_Person`
9. `UNI.Section` RT_{ext}`CS.Division`
10. `CS.Student` NT_{ext}`TP.Student`

Relationships from 1 to 4 and 9 are supplied by the designer; relationships 5 and 6 are intra–schema extensional relationships automatically extracted by the system from the is–a hierarchies of the `COMPUTER_SCIENCE` source; relationship 10 is inferred from 1 and 2.

3 Source Schema Matching

Goal of this phase of the integration process in ARTEMIS/MOMIS is to identify ODL_{I^3} classes candidate to integration, that is, classes that describe the same or semantically related information in different source schemas. To this end, *affinity coefficients* are evaluated for all possible pairs of ODL_{I^3} classes, based on the relationships in the Common Thesaurus properly strengthened. Affinity coefficients determine the degree of matching of two ODL_{I^3} classes c and c' based on their names (*Name Affinity* coefficient) and their attributes (*Structural Affinity* coefficient). A *Global Intensional Affinity* (GIA) coefficient is then determined as the linear combination of the Name and Structural Affinity coefficients. To evaluate such coefficients, an affinity function $A()$ is defined on top of the Common Thesaurus to evaluate the affinity of two terms. The affinity $A(t, t')$ of two terms t and t' is equal to the highest-strength path of terminological relationships (i.e., intensional) between them, if at least one path exist, and is zero otherwise. Given two terms t and t' and a path of terminological relationships between them, the strength of this path is computed by multiplying the strengths of all terminological relationships involved in it. $A(t, t')$ coincides with the strength of the highest-strength path between t and t', denoted by $\rightarrow^m$, that is, $A(t, t') = \sigma_{12_{\Re}} \cdot \sigma_{23_{\Re}} \cdot \ldots \cdot \sigma_{(m-1)m}$. To assess the level of affinity of two ODL_{I3} classes in a comprehensive way, a Global Affinity coefficient taking into account also the knowledge provided by extensional properties is introduced. We denote

Table 1. Affinity coefficients for schema matching

Coefficient	Definition	Description
$NA(c, c')$	$A(n_c, n_{c'})$	$NA(c, c')$ is the value of the affinity between the names of two classes, if this value exceeds a specified threshold.
$SA(c, c')$	$\frac{2 \cdot \lvert\{(a_t, a_q) \lvert a_t \in A(c), a_q \in A(c'), n_t \sim n_q\}\rvert}{\lvert A(c)\rvert + \lvert A(c')\rvert}$	$SA(c, c')$ is the measure of the level of structural matching of the classes, proportional to the number of their attributes whose names have affinity and whose domains are compatible.
$GIA(c, c')$	$w_{NA} \cdot NA(c, c') + w_{SA} \cdot SA(c, c')$	$GIA(c, c')$ is a global measure of the affinity of two classes based on intensional properties only, computed as the linear combination of their name and structural affinity, respectively.
$GA(c, c')$	$GIA(c, c')$+(1-$GIA(c, c')$) $\cdot\ \sigma_{\Re_{ext}}(c, c')$	$GA(c, c')$ is a comprehensive measure of the affinity of two classes, which takes into account also the extensional properties holding between the two classes.

Legend: $A(c)$ and $A(c')$ are the sets of attributes of c and c', respectively;
$\lvert X \rvert$ denotes the cardinality of set X; $\sim$ denotes name affinity

by $\Re_{ext}$, with $\Re_{ext} \in \{SYN_{ext}, BT_{ext}, NT_{ext}, RT_{ext}, DJS_{ext}\}$ an inter-schema extensional relationship. For affinity evaluation, each extensional relationship $\Re_{ext}$ is associated with a strength $\sigma_{\Re_{ext}} \in [0, 1]$, expressing its implication for affinity evaluation. Following what has been done in assessing the strengths of the terminological relationships, we consider equivalence (SYN_{ext}) as the strongest indicator for affinity; moreover we consider inclusion (BT_{ext}/NT_{ext}) stronger than overlapping (RT_{ext}). Table 1 summarizes the affinity coefficients.

Global affinity coefficients are then used by a hierarchical clustering algorithm, to classify ODL_{I^3} classes. The output of the clustering procedure is an affinity tree, where ODL_{I^3} classes are the leaves and intermediate nodes have an associated affinity value, holding for the classes in the corresponding cluster. Clusters for integration (candidate clusters) are interactively selected from the affinity tree using a threshold based mechanism.

Candidate clusters are characterized with respect to *size* and *level of homogeneity*. The size of a candidate cluster is the number of ODL_{I3} classes contained. The level of homogeneity of a candidate cluster is given by the threshold value, and consequently by the affinity value associated with the cluster in the affinity

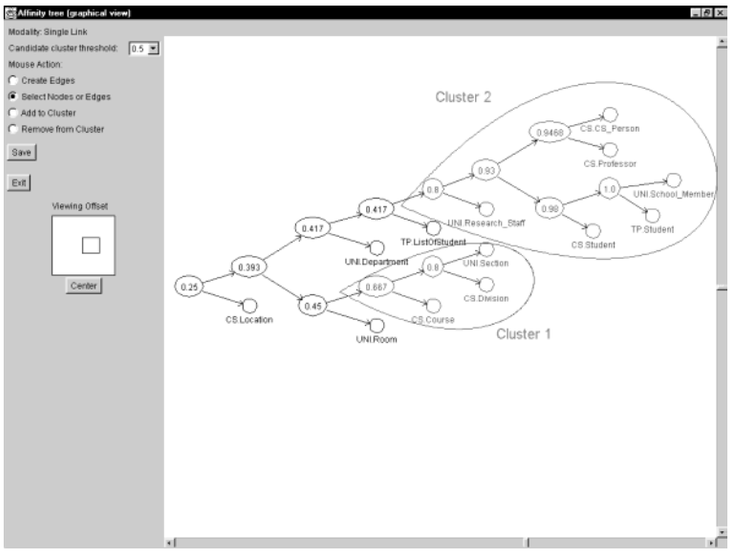


Fig. 2. Candidate clusters with extensional properties

tree (i.e., a high value of affinity corresponds to a high level of homogeneity). The number, size, and level of homogeneity of candidate clusters is determined by the designer who can set the most suitable value for the threshold γ.

In Figure 2, we show the affinity tree computed by considering both intensional and extensional relationships, using a threshold value $\gamma = 0.5$.

4 Global Schema and Mapping Generation

For each selected cluster in the tree, a global ODL$_{I^3}$ class providing the unified view of all the classes of the cluster is defined in ARTEMIS/MOMIS. The generation of global classes is interactive with the designer. Let Cl_i be a selected cluster in the affinity tree and *global _class*$_i$ the global ODL$_{I^3}$ class to be defined for Cl_i. First, we associate with *global _class*$_i$ a set of global attributes, corresponding to the union of the attributes of the classes belonging to Cl_i. The attribute unification process is performed automatically for what concerns names according to the following rules: i)for attributes that have a SYN relationship, only one term is selected as the name for the corresponding global attribute in *global _class*$_i$; ii)for attributes that have a BT/NT relationship, a name which is a broader term for all of them is selected and assigned to the corresponding global attribute in *global _class*$_i$.

To complete global class definition, information on attribute mappings and default values is provided by the designer in the form of *mapping rules*. For each global ODL$_{I^3}$ class a persistent *mapping-table* storing all the mappings is

University_Person	name	dept	e_mail	section	school	
UNI.Research_Staff	name	dept_code	e_mail	s_code	null	...
UNI.School_Member	name	null	e_mail	null	school	...
CS.CS_Person	first_name and last_name	null	null	null	"cs"	...
CS.Student	first_name and last_name	null	e_mail	null	"cs"	...
CS.Professor	first_name and last_name	null	null	null	"cs"	...
TP.Student	name	null	e_mail	null	school_name	...

	year	belong_to	takes	rank	s_code	tax_fee
...	null	null	null	"professor"	null	null
...	year	null	null	"student"	null	null
...	null	null	null	null	null	null
...	year	null	takes	rank	null	null
...	null	"belong_to"	null	rank	null	null
...	null	null	null	"student"	s_code	tax_fee

Fig. 3. Mapping Table for the global class `University_Person`

generated; it is a table whose columns represent the set of the local classes belonging to the cluster and whose rows represent the global attributes. An element $MT[L][ag]$ represents the set of attributes of the local class L which are mapped into the global attribute ag: the value of the ag attribute is a function of the values assumed by the set of attributes $MT[L][ag]$. Some simple and frequent cases of such function are the following (see Figure 3 as an example):

- *identity* : the (ag) value *is equal to* the la value; we denote this case as $MT[L][ag] = la$.
- *concatenation* : the ag value is obtained as a concatenation of the values assumed by a set of local attributes la_i of the local class L; we denote this case as $MT[L][ag] = la_1$ `and` $\ldots$ `and` la_n (see MT[`CS.Student`][`name`]).

When the ag has no correspondence with any attribute of the local class L, the possible choices are:

- *constant* : ag assumes into the local class L a costant value set by the designer; we denote this case by $MT[L][ag] =$ `const` (see the `rank` attribute).
- *undefined* : ag is set undefined into the local class L; we denote this case by $MT[L][ag] =$ `null`.

5 Semantic Query Optimization and Reformulation

The optimized query reformulation is obtained on the basis of the computed Base Extension set and by using the semantic query optimization techniques previously developed by the authors [1].

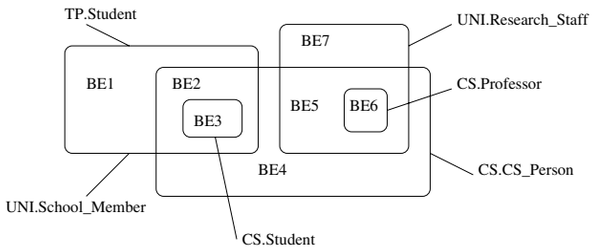


Fig. 4. Base extension Set for the global class University_Person

CL \ BE	BE1	BE2	BE3	BE4	BE5	BE6	BE7
UNI.School_Member	1	1	1	0	0	0	0
UNI.Research_Staff	0	0	0	0	1	1	1
CS.CS_Person	0	1	1	1	1	1	0
CS.Student	0	0	1	0	0	0	0
CS.Professor	0	0	0	0	0	1	0
TP.Student	1	1	1	0	0	0	0

Fig. 5. Tabular representation of the Base Extension Set of University_Person

5.1 Base Extensions

Intuitively, given a global class, a Base Extension gathers up all objects of some local classes such that the set of Base Extensions satisfies all the extensional relationships defined over the set of local classes and allows a partitioning of the set of the sources objects.

Definition 1 (Base Extension Set). *Let* $G = (\boldsymbol{L}_G, \boldsymbol{GA}, MT)$ *be a global class and* $\mathcal{I}$ *be an instance of the inter-sources schema* $\sigma\colon \boldsymbol{L} \rightarrow \mathbf{S}(\boldsymbol{LA}, \boldsymbol{L})$. *A* set of base extensions *of* G *on* $\mathcal{I}$ *is a pair* $(\boldsymbol{B}, F)$, *where* $\boldsymbol{B}$ *is a set of* base extension names *(denoted by* $B_1, B_2, \ldots$*),* $\mathcal{I}(B_i) = \bigcap_{L \in F(B)} \mathcal{I}(L)$, *and* F *is a total function* $F : \boldsymbol{B} \rightarrow 2^{\boldsymbol{L}_G}$ *such that:* $\bigcup_{B \in \boldsymbol{B}} F(B) = \boldsymbol{L}_G$, *and the set* $\left\{\mathcal{I}(B) - \bigcup_{L \in (\boldsymbol{L}_G - F(B))} \mathcal{I}(L) \mid B \in \boldsymbol{B}\right\}$ *is a partition of* $\bigcup_{L \in \boldsymbol{L}_G} \mathcal{I}(L)$.

At present, we adopt the algorithm of [12] to determine a base extension set[1].

Figure 4 shows the Base Extension Set for the Global Class `University_Person`. A Base extension set of a Global Class G is represented by a table. Table rows represent the local classes of the global class and table columns represent the base extensions. The presence of a `1` in the table cell (L, B) means $L \in F(B)$ (e.g. see Figure 5).

[1] More than one base extension set can be obtained on the basis of the above definition; the discussion about the quality of the selected set is out of the scope of this paper.

$A(BE1) = \{\texttt{name}, \texttt{year}, \texttt{school}, \underline{\texttt{rank}}, \texttt{e\_mail}, \texttt{s\_code}, \texttt{tax\_fee}\}$
$A(BE2) = \{\texttt{name}, \texttt{year}, \texttt{school}, \underline{\texttt{rank}}, \texttt{e\_mail}, \texttt{s\_code}, \texttt{tax\_fee}\}$
$A(BE3) = \{\texttt{name}, \texttt{year}, \texttt{school}, \texttt{rank}, \texttt{e\_mail}, \texttt{s\_code}, \texttt{tax\_fee}, \texttt{takes}\}$
$A(BE4) = \{\texttt{name}, \underline{\texttt{school}}\}$
$A(BE5) = \{\texttt{name}, \underline{\texttt{rank}}, \texttt{dept}, \texttt{e\_mail}, \texttt{section}, \underline{\texttt{school}}\}$
$A(BE6) = \{\texttt{name}, \texttt{rank}, \texttt{dept}, \texttt{e\_mail}, \texttt{section}, \texttt{belong\_to}, \underline{\texttt{school}}\}$
$A(BE7) = \{\texttt{name}, \underline{\texttt{rank}}, \texttt{dept}, \texttt{e\_mail}, \texttt{section}\}$

Fig. 6. Base Extension Attributes

The attributes of a Base Extension B are the global attributes which are mapped, by a not null mapping, into a local class of B. Formally:

Definition 2 (Base Extension Attributes). *Let $G = (\boldsymbol{L}_G, \boldsymbol{GA}, MT)$ be a global class and $(\boldsymbol{B}, F)$ be the set of base extensions of G, then the attributes of a base extension $B \in \boldsymbol{B}$ are defined as:*

$$A(B) = \{ag \in \boldsymbol{GA} \mid \exists L \in F(B), MT[L][ag] \not= \texttt{null}\}.$$

The Base Extension Attributes of our example are shown in Figure 6, where underlined attributes correspond to constant mapping.

Definition 3 (Domination). *Given a global class $G = (\boldsymbol{L}_G, \boldsymbol{GA}, MT)$, the set of base extensions $(\boldsymbol{B}, F)$ of G, and $B_1, B_2 \in \boldsymbol{B}$ we say that B_1* dominates (dom) *B_2 w.r.t. the set of global attributes $X \subseteq \boldsymbol{GA}$ iff $X \subseteq A(B_1) \cap A(B_2) \wedge F(B_1) \subset F(B_2)$.*

5.2 Query Plan and Execution

We will show the effectiveness of our optimization method by means of the following (mediated) query:

```
Q: select e_mail
    from University_Person
    where school = 'eng'
    and (s_code = 'a1x' or year = '2001' or belong_to = 'eng')
```

Processing the above query, without considering extensional knowledge, would individuate all the local classes for which at least one of the mediated query attributes has a not null mapping with it. The candidate local classes are thus all the local classes of `University_Person` but `UNI.Research_Staff` and `CS.CS_Person`. The query has thus to be reformulated on the basis of the above classes.

The approach is performed in the following steps.

1. Determination of the Local Class Set We consider the *Disjunctive Normal Form - DNF* of the query condition: $DNF =$ `F1 or F2 or F3` where:

`F1` = `(school='eng') and (s_code='a1x')`
`F2` = `(school='eng') and (belong_to='eng')`
`F3` = `(school='eng') and (year='2001')`

For each factor F of DNF we define the set: $BE(F) = \{B \mid \forall\ ag \text{ of } F, ag \in A(B)\}$, i.e., $B \in BE(F)$ iff $A(B)$ contains all the global attributes of the factor F. Then we define the set $BE_{min}(F) = \{B \mid B \in BE(F) \wedge \nexists\ B' \in BE(F) \mid B' dom\ B\}$.

In our example $BE($`F1`$) = BE($`F3`$) = \{BE1, BE2, BE3\}$ and $BE($`F2`$) = \emptyset$, since the involved attributes, i.e. `school` and `belong_to`, are contained in A(BE6), but the `school` is mapped to the constant value `'cs'` and thus `(school='eng')` is false.

Intuitively, a factor F of DNF such that $BE(F) = \emptyset$ can be eliminated as the value of F is always false. In our example, we obtain a simplified $DNF =$ `F1 or F3`. On the basis of this simplification, we obtain the following result: we do not have to access the local class `CS.Professor` as the only conjunctive predicate related to their attributes `(school='eng') and (belong_to='eng')` has been eliminated. Local classes are determined by considering the union of all the local classes included in $BE_{min}(F_i)$. With reference to our example, we have $BE_{min}($`F1`$) = BE_{min}($`F3`$) = \{BE1\}$, as $BE1$ dominates both $BE2$ and $BE3$; as a consequence the identified local classes are: {`TP.Student`, `UNI.School_Member`}. Notice that the classes `CS.Student` and `CS.CS_Person` are excluded from query execution too: they are useless as their objects are also instances of other local classes (as, for instance, stated by `CS.Student` NT_{Ext} `UNI.School_Member`) and their contributions to the query, that is attributes `school` and `e_mail`, are already provided by the identified local classes.

2. Query Reformulation For each pair of local classes belonging to the same base extension the related join attributes are considered. In our example, we have only a base extension, $BE1$, then the local classes are `TP.Student` and `UNI.School_Member` and the join attribute is `name` for both the classes.

Then, we consider the simplified DNF obtained in the previous phase and, for each factor F, we build a *local query* for each local class of $BE(F)$. The query `<condition>` is the is the conjunction of all predicates of the factor F which can be *solved* in the local class `L` (at least one predicate since `L`$\in BE(F)$) and the `<select-list>` is obtained by adding to the query select all the join attributes.

In our example, we have four local queries (a local query for each factor):

```
QF1L1: select e_mail, name      QF1L2: select e_mail, name
       from TP.Student                 from UNI.School_Member
       where (school_name='eng')       where (school = 'eng')
       and (s_code='a1x')

QF3L1: select e_mail, name      QF3L2: select e_mail, name
       from TP.Student                 from UNI.School_Member
```

```
       where (school_name='eng')        where (school = 'eng')
                                        and (year = 2001)
```

3. Local Query Execution For each source, by using ODB-Tools, we calculate the query inheritance hierarchy w.r.t. subsumption relationship and we send to the wrapper only the most generalized queries, enriched in the `<select-list>` with the local attributes contained into the other local queries, to be translated and executed by the local sources. This approach reduces the data access cost by the avoidance of redudant multiple accesses to the local sources. In the example we execute only `QF1L2` and `QF3L1`, rewrited with other local attributes:

```
QF3L1: select e_mail,name,s_code   QF1L2: select e_mail,name,year
       from TP.Student                     from UNI.School_Member
       where (school_name='eng')           where (school = 'eng')
```

`QF1L1` and `QF3L2` will be obtained at the mediator level on the basis of local queries results:

```
QF1L1: select e_mail, name      QF3L2: select e_mail, name
       from QF1L2                      from QF3L1
       where (s_code='a1x')            where (year = 2001)
```

4. Mediated Query Execution The first step of the Mediated Query Execution is the object fusion of the local query results belonging to the same base extension; this is implemented by a query, called *object fusion query*, which is performed for each factor and each base extension previously individuated.

In our example, for factor `F1` we have only the base extension $BE1$ and the object fusion of `QF1L1` and `QF1L2` is based on the direct–join on the join attribute `name`, thus we obtain the following object fusion query (the analogously for `F1`):

```
QF1BE1: select e_mail                QF3BE1: select e_mail
        from QF1L1,QF1L2             from QF3L1,QF3L2
        where QF1L1.name=QF1L2.name  where QF3L1.name=QF3L2.name
```

of course, all the object fusion queries have the same `<select-list>`.

The second and last step of the Mediated Query Execution is the full outer join of the object fusion queries: in our example, the full outer join between `QF1BE1` and `QF3BE1`.

6 Concluding Remarks

In this paper, we have described the semantic integration and query optimization process for heterogeneous data sources with reference to the ARTEMIS/MOMIS system. Main features of ARTEMIS/MOMIS can be summarized as follows: i) use of formalization and reasoning capabilities of Description Logics for a semantically rich representation of heterogeneous data sources for both semantic integration and query reformulation and optimization: ii) semiautomatic extraction

of interschema properties relating schema concepts of data sources at the conceptual level; iii) derivation of an integrated and unified representation of the involved data sources with explicit representation of the schema mappings and interschema knowledge exploited for query processing and semantic optimization.

References

[1] D. Beneventano, S. Bergamaschi, C. Sartori, and M. Vincini. ODB-QOPTIMIZER: A tool for semantic query optimization in OODB. In *Int. Conf. on Data Engineering - ICDE97*, 1997. http://sparc20.dsi.unimo.it. 156, 160

[2] S. Bergamaschi, S. Castano, D. Beneventano, and M. Vincini. Semantic integration of heterogenous information sources. *Data and Knowledge Engineering*, 36(3):215–249, 2001. 155

[3] S. Castano, V. De Antonellis, and S. De Capitani di Vimercati. Global viewing of heterogeneous data sources. *IEEE Transactions on Data and Knowledge Engineering*, 13(2), 2001. 155

[4] T. Catarci and M. Lenzerini. Representing and using interschema knowledge in cooperative information systems. *Journal of Intelligent and Cooperative Information Systems*, 2(4):375–398, 1993. 156

[5] S. Cluet, P. Veltri, and D. Vodislav. Views in a large scale XML repository. In *Proc. of the VLDB 2001*, Roma, Italy, 2001. 155

[6] A. Doan, P. Domingos, and A. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *Proc. of ACM SIGMOD*, Santa Barbara, California, USA, 2001. 155

[7] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In *Proc. of the Sixteenth ACM SIGMOD Symposium on Principles of Database Systems*, 1997. 155

[8] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the lore data model and query languages. In *Proc. of International Workshop on the Web and Databases (WebDB'99)*, pages 25–30, Philadelphia, Pennsylvania, USA, 1999. 154

[9] R. Hull. Managing semantic heterogeneity in databases: A theoretical perspective. In *Proc. of the 16th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS'97)*, 1997. 155

[10] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *Proc. of VLDB 2001*, Roma, Italy, 2001. 155

[11] R. Pottinger and A. Y. Levy. A scalable algorithm for answering queries using views. In *Proc. of VLDB 2000*, pages 484–495, Cairo, Egypt, 2000. 155

[12] I. Schmitt and C. Türker. An Incremental Approach to Schema Integration by Refining Extensional Relationships. In *Proc. of the ACM CIKM 98*, pages 322–330, New York, 1998. ACM Press. 155, 161

Extracting Information from Semi-structured Web Documents

Ajay Hemnani and Stephane Bressan

National University of Singapore
3 Science Drive 2, Singapore 117543
{hemnania,steph}@comp.nus.edu.sg

Abstract. The World Wide Web has now entered its mature age. It not only hosts and serves large amounts of pages but also offers large amounts of information potentially useful for individuals and businesses. Modern decision support can no more be effective without timely and accurate access to this unprecedented source of data. However, unlike in a database, the structure of data available on the Web is not known a priori and its understanding seems to require human intervention. Yet the conjunction of rules for interpreting layout and simple domain knowledge enables in many cases the automatic extraction of such data. In such cases we say that data is semi-structured. In this paper, we present a framework in which we try to address the problem of extracting semi-structured data. This framework combines a syntactical extraction strategy with a set of mapping rules, heuristics and simple domain knowledge, which maps a syntactical structure identified in Web documents to a conceptual/semantic structure. We present and analyse one instance of this framework in which a syntactical extraction strategy exploits the HTML structure of Web documents using a Tree Alignment algorithm with a novel combination of heuristics to detect repeated patterns and infer rules to extract relevant records. Then, by the use of domain knowledge, we refine the extraction rules such that not only are they able to extract data, but they also construe meaning to the extracted results.

1 Introduction

The World Wide Web has now entered its mature age. It not only hosts and serves large amounts of documents but also offers large amounts of information potentially useful for individuals and businesses. Web documents can be modeled using the three ingredients: *format*, *content*, and *structure* [5, 11]. *Format* is the visualized view of a document. *Content* is the actual data that a document has. *Structure* is the logical organization of a document. The structure defines the relationships of the elements as well as the order in which the elements are assembled. However, there is significant diversity among these documents with respect to structure as well as content. Hsu [10] suggests the following categorization of Web documents: A Web document that provides itemised information is structured if each attribute in a tuple can be correctly extracted based

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 166–175, 2002.

on some uniform syntactic clues, such as delimiters or the orders of attributes. Semi-structured Web documents, however, may contain tuples with missing attributes, attributes with multiple values, variant attribute permutations, and exceptions. A Web document is unstructured if linguistic knowledge is required to extract attributes correctly.

Structured information can easily be correctly extracted using the strict format description. Moreover, a significant portion of the data on the Web is semi-structured. Examples of such documents include online classifieds, product catalogs, staff directories, search engine listings, and so on. Hence, we focus on extracting information from these semi-structured Web documents. Manual parsing of these documents is tedious, time-consuming and unnecessary. Furthermore, most of these sources often do not exhibit rich grammatical structure, and neither do they contain complete sentences. Therefore, traditional NLP techniques are not suitable for data extraction. For these semi-structured Web documents, the data is usually organized as sets of record-size chunks and so the regularity of their appearance can be exploited for extracting data instead of using linguistic knowledge. This logical organization of data is reflected by the structure of the document. And this structure is represented using some syntax, comprising of HTML tags. Hence, these structural components can be used as extraction targets. This has motivated a lot of research efforts concerning the invention of information agents that automatically extract information from multiple websites. These agents rely on a set of extraction rules, and are more commonly known as wrappers. Since most wrappers are purely syntax-based, extraction procedures based on such wrappers only succeed in extracting data from Web documents, but fail to sieve out their semantics. This renders the extraction procedure as incomplete. On the other hand, our aim is to design wrappers that know *what* to extract, *how* to extract, and *show* how the extracted data items relate to each other.

We believe that within the structural arrangement identified in Web documents, there lies some implicit semantic information about the content. In order to construe this hidden semantics to the content, we rely on domain knowledge. Based on this idea, we form a framework (see Figure 1) for extracting information from Web documents. This framework provides the groundwork for designing wrappers. We first develop a wrapper using syntactical clues found within the Web document. Then, using simple domain knowledge, we fine-tune the wrapper to incorporate semantical understanding of its target data. We explain the internal mechanisms of our framework with the following example. Consider a simple Web document of a bookstore, listing the catalog in a tabular format. Let's assume that this table has 3 columns, displaying the book title, author and price information, in this particular order. Let's further assume that each record is enclosed within HTML tags according to the following syntax:

<TR><TD>title</TD><TD>author</TD><TD>price</TD></TR>.

This could be seen as one complete book record, and the 3 columns represent the 3 attributes of the book. Now, the complete extraction task for this document would comprise the following 3 steps: (1) identify the syntactical structure of the table within this Web document that contains the book records (2) convert this syntactical structure to a conceptual/semantic structure such that we know what each of the columns inside the table entails, and how they relate to each other (3) extract all book records from the table and store them as tuples in the database.

In most Web documents, data is usually presented together with menus, contact information and other details that normally appear in headers and footers of documents. Hence, for step 1, we need to identify the relevant section of the document and then extract all the record-size data chunks. For this purpose, we propose a novel approach that uses a Tree Alignment algorithm and a set of

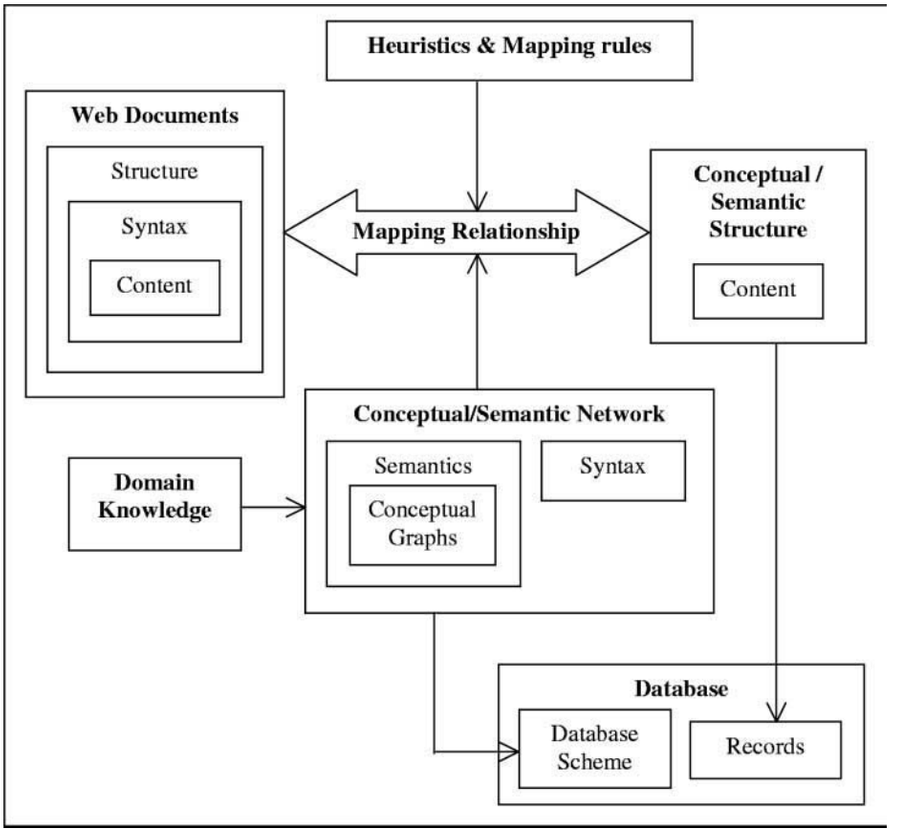


Fig. 1. The proposed framweork

heuristics to detect repeated patterns and infer rules to extract relevant records. This approach is purely syntax-based. Continuing with the above example, after performing step 1 successfully, we would have identified the relevant portion of the Web document in question, formed a generalized extraction rule and extracted all data records with it. The generalized extraction pattern (or rule) for this example would be:

<TR><TD>Text</TD><TD>Text</TD><TD>Text</TD></TR>.

For step 2, we rely on domain knowledge. For this example, the domain knowledge would comprise information about books and their attributes. For instance, a book usually has a title, author, price, version, and so on. In turn, the price information itself has its own attributes like the amount and currency. All these details are modeled as separate entities or concepts along with their relationships using the conceptual graph theory [17]. Also, each concept will have a syntax in which it is represented. Having defined the semantics as well as the syntax of all the concepts involved, we form a complete *conceptual/semantic network* for that particular domain. The schema for the dataset is deduced from this conceptual/semantic network and all records will be stored based on this schema. Finding a mapping relationship between the conceptual network and the given syntactical structure of the records (as depicted by the extraction rule) completes step 2. This mapping relationship explains the contents of the records at the data field level. In the final step (step 3), the mapping relationship is used to convert the records to conform to the schema and these records are stored as tuples in the database.

2 Related Work

Recent research efforts on structuring Web information have mainly focused on wrapper generation, natural language processing and Web data modelling. Automatic wrapper generation uses machine-learning techniques, and the wrapper research community has developed learning algorithms for a spectrum of wrappers from the simplest to the most sophisticated. Systems like RAPIER [2], SRV [8] and WHISK [16] are examples of wrapper generators based on techniques intended for Web documents with a less rigorous structure. Their focus is to develop machine-learning methods for the Information Extraction problem. The systems must usually go through a training phase, where they are fed with training examples that have to be manually labeled. This poses a question of economy of scale.

Since the mid-nineties, the DB research community has also been working on the same problem. In the Araneus project [1], different languages are used for extracting data and defining database views of the Web. TSIMMIS [4] uses OEM (Object Exchange Model) as a common data model. Other systems like ShopBot [6], WIEN [13], SoftMealy [10], and STALKER [15] use techniques

that are tailored for relatively well-structured Web documents. Embley et al. [7] adopt a heuristic approach to discover record boundary in multiple-record Web documents. However, this one-tag separator approach fails when the separator tag is used elsewhere among a record other than the boundary.

Continuing with the approach of analyzing structural content of Web documents, Lakshmi [14] proposes extracting information from the Web through exploiting the latent information given by HTML tags. For each specific extraction task, an object model is created consisting of the salient fields to be extracted and the corresponding extraction rules based on a library of HTML parsing functions. Chang et. al. [3] built a system that can automatically identify the record boundary by repeated pattern mining and multiple sequence alignment using a data structure called PAT trees to help in the discovery of repeated patterns. The main application of PAT trees is in the domain of exact matching. To allow approximate matching, the technique for multiple string alignment is adopted to extend the discovered exact repeats.

3 Extracting Information from Web Documents

This section presents our syntax-based extraction approach that is based on the Tree Alignment algorithm with a combination of heuristics to detect repeated patterns and infer rules to extract relevant records. Our approach has been published in [9]. Just to recapitulate, here we briefly summarize the proposed approach.

3.1 Syntax-Based Extraction Approach

We first read in a Web document D and build a tag tree T, based on the algorithm used by Embley et al. [7]. Next, we locate the subtree S within T that contains the records of interest using the enhanced highest fan-out rule. Once the subtree S has been identified, the next task would be to discover the record boundary and detect repeating patterns of tags that embed these records, and align them, using the Tree Alignment algorithm, to form generalized extraction rules that will cater to all the records in the Web document. The idea of alignment of trees was proposed by Jiang et. al. [12], and used for approximate tree matching to compute a measure of similarity between 2 ordered, labeled trees. The idea of alignment of trees can be easily generalized to more than 2 trees. Although it is very compelling to believe that each subtree under S is possibly a record by itself, it is not always the case with many of the Web documents. Hence, it is necessary to detect the boundary of these records, i.e. how many of the consecutive subtrees under S actually make up one record. We reduce the problem of record boundary discovery to that of finding how many consecutive subtrees does a single record span. We define two heuristics, which we use in conjunction with the multiple tree alignment algorithm to cluster "similar" patterns together and align all the patterns in each of the clusters independently.

3.2 Performance Study of the Approach

We tested the performance of our approach on two categories of Web documents: Search engines and Classifieds. We use performance metrics such as recall and precision rates to analyze the performance of our approach. *Recall rate* is defined as the ratio of number of records of interest extracted by the extraction rule to the number of records of interest contained in the input Web document. *Precision rate* is defined as number of records of interest extracted by the extraction rule to the total number of records extracted. We conducted the experiments on 2 encoding levels: (1) all tags are taken into consideration (2) text-enhancement tags (<B>, <I>, <U>) are ignored.

The experimental results (in tables 1 and 2) confirm that the proposed heuristics-based approach improves the precision rate to a considerable extent. The recall rate, on the other hand, improves when text-enhancement tags are ignored. The results also indicate that the encoding level does not have any effect on the precision rate. Encoding level 2 is the better of the two schemes. Under encoding level 2, our proposed heuristics-based approach has an average recall rate of 97.2% and an average precision rate of 97.6% for documents from search engines. As for online classifieds, recall rate averages to 99.6% and precision rate averages to 98%.

Table 1. Experimental results for Search Engines

Websites	**Encoding (1)**		**Encoding (2)**	
	Recall	**Precision**	**Recall**	**Precision**
www.yahoo.com	55%	97%	91%	97%
www.webcrawler.com	100%	100%	100%	100%
www.lycos.com	60%	98%	95%	98%
www.infoseek.com	100%	97%	100%	97%
www.savvysearch.com	100%	96%	100%	96%

Table 2. Experimental results for Online Classifieds

Websites	**Encoding (1 & 2)**	
	Recall	**Precision**
www.asiaone.com.sg	100%	97%
www.classifieds2000.com	100%	95%
www.catcha.com	100%	100%
www.asiaxpat.com.sg	98%	98%
www.herald.ns.ca/classifieds	100%	100%

4 Using Domain Knowledge

This section describes the way we represent and use domain knowledge to make sense of the extracted data. We illustrate its application through an example.

4.1 Representing Domain Knowledge

We borrow the idea of conceptual graphs [17] to represent the acquired knowledge of a particular domain. To cater to the needs of our approach, we incorporated minor modifications into their representation, but the basic idea remains the same. Using the book catalog example we gave in section 1, we would have the following conceptual graph (CG) in the Display Form (DF) (see Figure 2).

This graph depicts the following: *a certain author wrote a certain book that is selling at a certain price*. In the display form, concepts are represented by rectangles. For e.g. the concept [Author] represents an instance of an author. Circles represent conceptual relations. For e.g. the conceptual relation (Write) relates an author to a book. Arrows represent the arcs that link the relations to the concepts: the first arc has an arrow pointing toward the relation, and the second arc has an arrow pointing away from the relation. If a relation has more than two arcs, the arcs are numbered. The linear form (LF) for this graph is given as follows: [Author] → (Write) → [Book] → (Sell) → [Price]. In the linear form, concepts are represented by square brackets instead of boxes, and the conceptual relations are represented by parentheses instead of circles. The display form represents the abstract CG most directly.

The above CG is still not complete, because each concept has not reached its lowest level yet. For instance, the concept [Price] may have attributes like the **amount** and **currency**. Now, amount is at the lowest level of detail. It is basically a number, and can be represented by #PCDATA. This piece of information has syntax as well. For instance, amount could be a number from 0 to 999999 (6 digits). This could be represented using regular expressions. In the case of currency, we use a predefined list of accepted currencies and a generic syntax is used to express them.

Furthermore, the task of creating a database schema from the conceptual graph is very straightforward. The relations in the CG pass off as database relations. For instance, the relation *Writes* stores records of authors and the books they have written. The relation *Sell* stores records of books and their prices. Of course, further re-modeling needs to be done if the *Normal Form* of the database

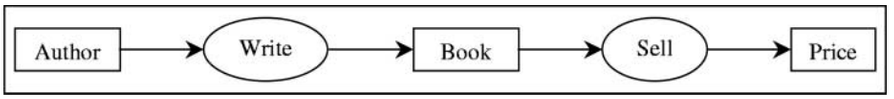


Fig. 2. Display Form of the Conceptual Graph for the book catalog domain

relations is of primary concern. Now, we introduce the notion of an *instance identifier* for a concept. To illustrate with an example, consider the concept Author. Say, famous writer "Sidney Sheldon" is instance of this concept. We call the data, "Sidney Sheldon", the instance identifier. The standard conceptual graph theory does employ this notion, but it does not explicitly specify them when describing the concept. Since we use the domain knowledge to create our database schema, it is imperative that we make this detail explicit. These identifiers could be used as *candidate keys* in schema generation for database relations. For e.g., amount is actually the instance identifier for the concept [Price], leaving it with only one attribute, the currency. With this notion, the Display Form of the modified CG for the book catalog domain is shown in Figure 3.

In this DF, each concept is accompanied by its instance identifier (represented by the word in italics following the concept name). Having said this, we need to mention that the concept [Currency] does not have an instance identifier because it is an *attribute* of another concept, namely the concept [Price]. Furthermore, the rule of the thumb is that *Attr* relations are not included as separate database relations in the schema. Instead, they co-exist within other relations. In this example, the *Sell* relation houses the book and price information (including the currency). We name attribute-based concepts (for e.g. currency) *secondary concepts*. All other concepts are *primary concepts*. For a particular domain, a complete set of concepts and relations form what we term as a *conceptual/semantic network*. Furthermore, each primary concept in this network represents a *semantic group*, comprising of itself and its related secondary concepts, if any.

4.2 Applying Domain Knowledge

Having described how we can model the domain knowledge as a conceptual / semantic network, we now show how to apply this knowledge to map the identified syntactical structure identified in a Web document to a conceptual/semantic structure. In essence, our approach is a 2-phase process: (1) the syntactical extraction mechanism extracts record-size data chunks embedded within a syntactical structure (2) convert the syntactical structure to the conceptual/semantic structure obtained from domain knowledge to make sense of the extracted data

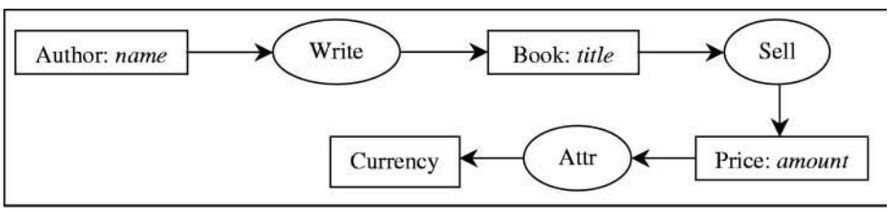


Fig. 3. Display Form of the modified Conceptual Graph for the book catalog domain

items. We illustrate this with the book catalog example from section 1. The generalized extraction rule formed in phase 1 has the syntax:
<TR><TD>$data_1$</TD><TD>$data_2$</TD><TD>$data_3$</TD></TR>.

Now, $data_1$, $data_2$ and $data_3$ are placeholders for the data items. Although we have extracted the data items, we have no idea what they mean. This is where domain knowledge plays its role. What we need to do here is analyze the data items, based on some semantic information, such that it would help construe meaning to these items. In this particular instance, the obvious task is to identify a mapping relationship between the 3 semantic groups, **book**, **author**, **price**, and the 3 placeholders. This reduces to a relatively easy task of matching the data items in those placeholders with the syntax (represented by regular expressions) of the identified concepts. Based on the highest number of matches, we can conclude which placeholder represents which semantic group. We follow a simple rule: *a placeholder p represents a semantic group g if the number of matches between p and g exceeds the number of matches between g and each of the remaining placeholders*. In ideal cases, we need only one Web document to obtain the mapping relationship and every placeholder will represent exactly one semantic group. In case of conflicts, a second Web document could be used to introduce fresh evidential support. For our example, the placeholder $data_1$ represents the [Book] concept; $data_2$ represents the [Author] concept and $data_3$ represents the [Price] concept. Hence, the generalized extraction rule is converted to:
<TR><TD>book</TD><TD>author</TD><TD>price</TD></TR>.

This improved rule tells more than just how to extract the relevant data. This rule knows what the data items in the respective placeholders mean. Since the schema is based on the concepts from the conceptual/semantic network, storing the extracted records becomes a very straightforward task once the mapping relationship between the placeholders and the semantic groups (which comprises of the concepts) has been identified.

5 Conclusion and Future Work

This paper discusses a framework for designing wrappers. In one instance of this general framework, we present and analyze a syntactical extraction technique that utilizes the HTML structure of the documents to discover repeated patterns, using a Tree Alignment algorithm with a set of heuristics, and forms extraction rules. The adoption of the Tree Alignment algorithm to repeated pattern discovery is very suitable since multiple sequence alignment is handled implicitly. As shown by our empirical performance evaluation on several Web sources, the efficiency (time) is practical and the effectiveness (recall and precision) reasonable. With the use of domain knowledge, the extraction rules are refined and become *intelligent* such that they are able to understand the semantics of their target data. This work is the first step towards the design of a more

complete solution in which syntactic and semantic knowledge can be combined and cross-leveraged for the automatic extraction of Web data.

References

[1] Atzeni, P., Mecca, G., Merialdo, P.: To Weave the Web. In Proc. Twenty-third International Conference on Very Large Data Bases (1997) 206–215 169

[2] Califf, M. E., Mooney, R. J.: Relational Learning of Pattern-Match Rules for Information Extraction. Working papers of the ACL-97 workshop in Natural Language Learning (1997) 169

[3] Chang, C. H., Lui, S. C.: Information Extraction Based on Pattern Discovery. In Proc. 10th International World Wide Web conference on World Wide Web (2001) 170

[4] Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., Widom, J.: The TSIMMIS project: integration of heterogeneous information sources. IPSJ Conference (1994) 7–18 169

[5] Colby, M., Jackson, D. S.: Using SGML. Que Corporation, Indianapolis, USA. Special edition (1996) 166

[6] Doorenbos, R. B., Etzioni, O., Weld, D. S.: A scalable comparison-shopping agent for the World Wide Web. In Proc. 1st international conference on Autonomous Agents. ACM Press., New York (1997) 39–48 169

[7] Embley, D., Jiang, Y., and Ng, Y. -K.: Record-boundary discovery in Web documents. In Proc. ACM SIGMOD International Conference on Management of Data. Philadelphia, Pennsylvania, (1999) 467–478 170

[8] Freitag, D. Information Extraction from HTML: Application of a general Machine Learning Approach. In Proc. 15th National Conference on Artificial Intelligence (1998) 169

[9] Hemnani, A., Bressan, S.: Information Extraction - Tree Alignment Approach to Pattern discovery in Web documents. In Proc. Thirteenth International Conference on Database and Expert Systems Applications (2002) (to appear) 170

[10] Hsu, C.-H., Dung, M.-T.: Generating finite-state transducers for semi-structured data extraction from the Web. Journal of Information Systems, 23(8) (1998) 521–538. 166, 169

[11] Hsu, J. Y., and Yih, W. T.: Template-based information mining from html documents. In AAAI 97. AAAI Press, August (1997) 166

[12] Jiang, T., Wang L., Zhang, K.: Alignment of trees - an alternative to tree edit. Combinatorial Pattern Matching (1994) 75–86 170

[13] Kushmerick, N., Weld, D., Doorenbos, R.: Wrapper induction for information extraction. In Proc. 15th International Joint Conference on Artificial Intelligence (1997) 169

[14] Lakshmi, V.: Web structure Analysis for Information Mining. PhD Dissertation, National University of Singapore (2001) 170

[15] Muslea, I., Minton, S., Knoblock, C.: A hierarchical approach to wrapper induction. In Proc. 3rd International Conference on Autonomous Agents (1999) 169

[16] Soderland, S.: Learning Information Extraction Rules for Semi-structured and Free Text. Machine Learning, vol. 34 (1999) 233–272 169

[17] Sowa, J. F.: Conceptual Graphs. NCITS.T2/98-003 (1998) 169, 172

Object-Oriented Mediator Queries to Internet Search Engines

Timour Katchaounov, Tore Risch, and Simon Zürcher

Uppsala Database Laboratory, Department of Information Technology
Uppsala University, Sweden

Abstract. A system is described where multiple Internet search engines (ISEs), e.g. Alta Vista or Google, are accessed from an Object-Relational mediator database system. The system makes it possible to express object-oriented (OO) queries to different ISEs in terms of a high level OO schema, the *ISE schema*. The OO ISE schema combined with the mediator database system provides a natural and extensible mechanism in which to express queries and OO views that combine data from several ISEs with data from other data sources (e.g. relational databases). High-level OO web queries are translated through query rewrite rules to specific search expressions sent to one or several wrapped ISEs. A generic ISE query function sends the translated queries to a wrapped ISE. The result of an ISE query is delivered as a stream of semantically enriched objects in terms of the ISE schema. The system leverages publicly available *wrapper toolkits* that facilitate extraction of structured data from web sources, and it is independent of the actual wrapper toolkit used. One such wrapper toolkit was used for generating HTML wrappers for a few well-known ISEs.

1 Introduction

To facilitate the combined access to data on the web with data from other databases, a system called ORWISE (Object-Relational Wrapper of Internet Search Engines) has been developed that can process queries combining data from different Internet search engines (ISEs) with data from regular databases and other data sources. The design of ORWISE leverages available *wrapper toolkits* to extract information from web pages. ORWISE has been implemented for three well-known search engines using a publicly available wrapper toolkit [31].

ORWISE is an extension to the database system Amos II [29,30], that is based on the *wrapper-mediator approach* [34] for heterogeneous data integration. The core of Amos II is an extensible object-relational database engine having mediation primitives in a query language *AmosQL* similar to the OO parts of SQL-99 and ORWISE thus permits SQL-99 like queries that combine ISE results with data from other types of sources such as relational databases [10] and XML [23]. Amos II is suitable for collecting and processing results from ISEs because its purpose is to act

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 176-186, 2002.

as a fast mediator database which can manage meta-data of heterogeneous and distributed data sources and efficiently process queries to the sources.

The generalized *ISE wrapper manager* ORWISE, described in this paper, makes it possible to easily access one or several ISEs from Amos II using different *ISE wrappers* for each engine. Combined with OO mediation facilities [4,17], it allows to process OO database queries that combine data from several ISEs with data from conventional databases and other data sources. In difference to relational systems for web queries [14], the data produced by ORWISE is not just text strings but much more semantically rich object structures in terms of an OO schema for ORWISE, called the *ISE schema* (Internet search engine schema). The ISE schema describes capabilities and other properties of the search engines along with the structure of their results.

ISEs have some special problems compared to 'conventional' databases:

- *Semi-structured interfaces*: There are no standard interfaces to ISEs such as ODBC and JDBC. Web forms are used for specifying queries and other inputs to them. The result of an ISE query is a semi-structured web document containing not just the query result but also auxiliary text, banners, etc., which need to be filtered out from the query result.
- *Query languages*: ISEs do not have a standardized query language such as SQL but every ISE has its own query language with varying syntax and semantics.
- *Autonomy*: The content, structure and availability are totally controlled by the information supplier.
- *Evolution*: Internet sites tend to change very often. A system that accesses a site has to be very flexible.
- *Heterogeneity*: The data delivered by ISEs have varying structures and the system has to reconcile semantic differences.

In order to handle the above problems we need reliable and flexible interfaces to the ISEs, here termed *ISE wrappers,* which can programmatically fill and submit web forms and parse the structure of an ISE result document searching for predefined patterns. An ISE wrapper must be flexible enough to cope with small changes in the web sites.

To specify web source wrappers ORWISE utilizes wrapper toolkits to extract useful information from web pages. ORWISE is designed to be independent of the actual wrapper toolkits used. We investigated several of them to make sure that the system works with all of them. For our first implementation we chose W4F [31] to generate ISE wrappers for three search engines - Google (http://www.google.com), AltaVista (http://www.altavista.com), and Cora Research Paper Search (Cora) (http://cora.whizbang.com).

The ISE wrappers are connected to the system through a generic query language function called *orwise*, which is a foreign function (implemented in Java) overloaded for each search engine. It returns objects of an ISE specific type[1] that describes the retrieved query results. New ISE wrappers can dynamically be added to the system by creating a new subtype of the system type *SearchEngine* for each new ISE and then

[1] We use the terms 'type' and 'class' as synonyms.

implementing some code (in Java) to interface its ISE wrapper. The overloading of the function *orwise* is used for facilitating the plug-in of new ISE wrappers.

Once a new search engine is connected to the *orwise* function it can be used in OO queries. Since the parameters for each implementation of *orwise* are search engine specific, such queries will be rather detailed with search engine specific parameters for, e.g., query strings, site names, etc. The system therefore provides high-level query functions that can be used for any ISE and where queries are specified uniformly. For example, the function *webSearch* is defined for every search engine to specify OO queries to it in a search engine independent form. The high-level OO query expressions need to be transformed before the actual call(s) to *orwise* is issued. The approach in Amos II is to implement a *translator* module for each kind of data source (search engine, relational database, etc.). In the case of ISEs, the translators rewrite the high-level query into search engine query specifications containing calls to *orwise*. Since different search engines have different ways of specifying searches, they have different rewrite rules.

In summary, ORWISE provides i) the ISE schema for describing and querying data from any ISE, ii) a mechanism to specify search engine specific translators, and iii) facilities to allow different wrapper toolkits to be easily plugged into the system.

2 Related Work

Many projects (e.g. [11,16,21,27,33]) use the mediator approach to data integration in general. The work presented here describes how an object-relational mediation framework [29] leverages upon an available wrapper toolkit to provide access to ISEs.

The use of object-relational approach in querying the structure of XML Web documents has been done, e.g., in [3,8,12,23]. A query language standard for XML, XQuery [35], is being developed with which the contents of XML documents can be queried and new XML documents constructed. All major ISEs use HTML, not XML. General Web query languages for HTML are proposed in [19,25]. These are general languages for querying well-formed Web documents and not directly suitable for defining embedded interfaces to ISEs.

By contrast *wrapper toolkits* [9,13,15,18,20,22,24,31] specify programmatic interfaces to web sources handling both sending commands and extracting structured data from responses. They often include some advanced pattern matching language to extract data from Web documents as regular expressions operating on varying levels of granularity. With a wrapper toolkit a web source wrapper is defined by processing *wrapper specifications*, consisting of statements to connect to web sources and to detect the parts of the text to be extracted. They allow new wrappers to be specified much easier than with manual programming and the developers need not master a complex query language. A good overview of projects related to wrapper construction for Web sources can be found in [31].

A wrapper toolkit can be a *wrapper-generator* that generates code (e.g. Java) implementing a web source wrapper [1,2,24,31]. It can also be a *wrapper-interpreter* where the web source wrapper is specified as commands, which are interpreted at run time [18,15]. ORWISE is designed to work with both wrapper-generators and wrapper-interpreters. Web source wrappers represent data differently and are not

sufficient themselves to combine data from Web sources and conventional databases. Therefore there is need for data mediation facilities along the lines of this paper.

In [26] it is shown that an OO query language indeed is very useful for specifying queries to text engines. Our work differs in that we propose leveraging upon using external wrapper toolkits, OO query rewrites, and the ISE schema. Furthermore, we explicitly model the capabilities of the search engines in the ISE schema, rather than in the internals of the system. The WSQ/DSQ [14] project proposes an architecture where web searches are specified as SQL queries to two virtual relational tables. Their relational tables are inflexible for the purpose, compared to our ISE schema. The focus of the work in [5] is re-write rules and cost models for integrating text search with other queries. Those rewrite rules are applicable in our translator too.

To the best of our knowledge, no other project proposes a system that uses inheritance and overloading to model ISEs and their results on the conceptual level, while at the same time the implementation is independent of, and leverages existing wrapper toolkits. Another major difference to other projects is that our object-oriented ISE schema distinguishes between the search engine specific descriptions of documents and the actual documents. Furthermore, the ISE capabilities are modeled in the ISE schema too.

3 Scenario

We have implemented the scenario of Figure 1 to illustrate the functionality of the system. In the scenario, an Amos II mediator is used to process queries that combine data from a relational DB2 database through ODBC with three ISEs, AltaVista, Google and Cora. The access to the three Internet search engines uses the ORWISE wrapper, while the relational database is accessed through an ODBC-wrapper.

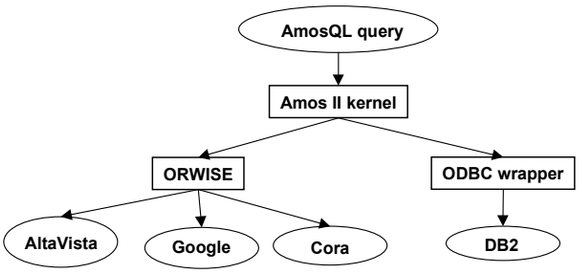


Fig. 1. Mediator scenario

The relational database stores a table of employees that is mapped to the mediator type *Employee*, using the techniques for defining OO views of relational databases [10]. The following AmosQL query uses Google to find the names of those employees who are mentioned in some web page in the web site `'www.csd.uu.se'`:

```
SELECT DISTINCT given_name(e), family_name(e)
FROM Employee e, DocumentView d, Google ise
WHERE d = webSearch(ise, given_name(e)) AND
      d = webSearch(ise, family_name(e)) AND
      host(url(d))=' www.csd.uu.se');
```

The first two lines of the 'where' clause in the query retrieve the documents that contain given names and family names of employees in the relational database, while the last line restricts the search to only those persons whose names are found by Google in web pages on the host `'www.csd.uu.se'`. Other text-related predicates such as `'near'` can also be added to refine the search. The type *DocumentView* represents descriptions of documents returned by an ISE and the type *Google* represents the wrapper for Google. The same query can be specified for Alta Vista by replacing the type *Google* with *AltaVista*. It is also possible to specify queries over several search engines by using the generic supertype *SearchEngine* instead of *AltaVista* or *Google*.

4 The ISE Schema

Queries to ISEs are posed in terms of the OO database schema on Figure 2. Inheritance and overloading are used to model heterogeneity of both ISEs and their results. Furthermore, we separate the description of results returned by ISEs from the documents themselves. Since Amos II has a functional data model [32], both type attributes and relationships between types are modeled by functions shown as think lines on Figure 2. For clarity, the overloaded function *orwise* is represented as an attribute of the subtypes of type *SearchEngine*. The core of the ISE Schema consists of three base types:

- *SearchEngine* – this type is used to categorize ISEs. It reflects the fact that search engines have different query capabilities and parameters. It has a subtype for each specific ISE normally with only one instance. The generic function *orwise* is overridden for each ISE to reflect their different semantics. Analogously each of them has a specific query rewrite function.
- *DocumentView* – objects of this type describe the results of a query to different ISEs. By introducing this type of objects we can distinguish between the documents themselves and the description of a document by an ISE. Document views often contain information about a document that is not part of the document itself and is imprecise or outdated. They may use different formats from the document itself; e.g. the Cora ISE returns HTML descriptors of PostScript documents. Differentiating between documents and views over documents allows for more precise queries.
- *Document* – describes document objects themselves. Subtypes of *Document* may describe document objects with different structure. The problem of querying structured documents is outside the scope of this work and has been addressed by other researchers [6], [28]. All this work can be easily reused in our system due to the flexibility of our OO data model.

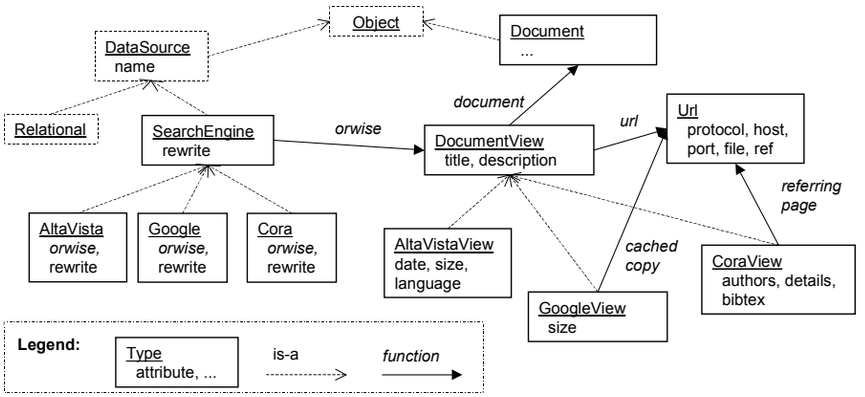


Fig. 2. The ISE schema

The two classes *DataSource* and *Relational* are part of the general Amos II meta-type hierarchy. The type *DataSource* serves as the base type of all meta-types for different kinds of data sources accessible through the mediator system. One such meta-type is *Relational*, which describes relational data sources. It has the function *sql*, analogous to the *orwise* function of *SearchEngine*. In our current implementation, the type *SearchEngine* has three subtypes for each of the wrapped search engines AltaVista, Google and Cora. Each of them defines its own version of the *orwise* function and specific rewrite rules. Correspondingly the type *DocumentView* has three subtypes: *AltaVistaView, GoogleView and CoraView*, where each of them has additional properties. For example, of the three ISEs only AltaVista returns the language of a document, while only Google may provide a locally cached copy of it's indexed documents, accessible through the function *cached_copy*. Finally, *Document* objects may be accessed and queried further through the *document* function of the type *DocumentView*. The type *Url* is an example of semantic enrichment of the ISE query results, as they return URLs as strings.

5 The ORWISE Architecture

Figure 3 shows the layered architecture of the system. The left part shows how ORWISE is interfaced with the Amos II kernel, while the right part shows the layers of ORWISE itself.

The architecture is designed to fulfill several requirements:

- It provides a uniform interface from the Amos II query processor to any ISE.
- It can use any existing general wrapper toolkits.
- It is independent of the wrapper toolkits used.
- It is possible to easily add a new ISE wrapper without any changes to the rest of the system.
- There is no need to modify the definitions of wrappers generated by wrapper-generators.

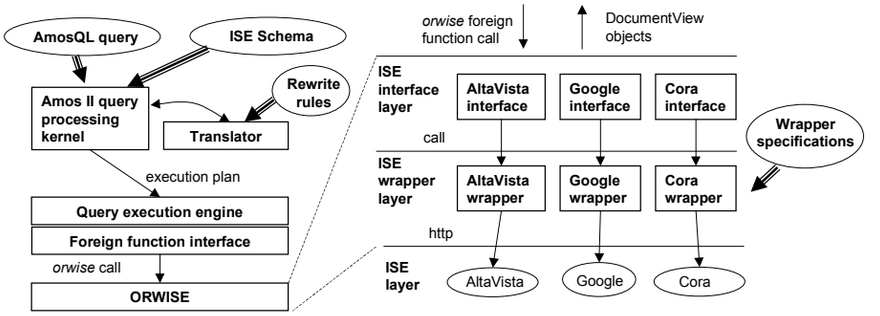


Fig. 3. System architecture

The two layers *ISE interface* and *ISE wrapper* fulfill these requirements. This architecture permits any wrapper toolkit to be used and different kinds of wrapper toolkits can even be combined.

The ISE interface layer defines an interface between the Amos II kernel and the underlying ISE wrapper layer used for interfacing each search engine. The functionality common for every ISE wrapper, such as instantiating ISE specific *DocumentView* objects and emitting the result stream, is encapsulated in this layer. It is called by the query processor and it calls the ISE wrapper for the chosen search engine. The basic foreign function interface of Amos II allows new ISE interfaces to be dynamically added to a running system. The ISE wrappers are specified by some external wrapper toolkit(s) chosen for each particular search engine. Therefore, the functionality they expose can be very different and cannot be directly used by ORWISE. The ISE interface therefore must instantiate objects, convert strings to URL objects or numbers, etc.

The ISE wrapper layer consists of the modules specified through the wrapper toolkit. It forms and sends HTTP requests to an ISE server and then extracts the results from the so received HTML page. The input to a wrapper toolkit is a specification of request submission and data extraction rules for a web source. The chosen W4F [31] toolkit is a wrapper-generator, which generates Java classes per wrapped data source. In this case the layer consists of the generated code. For wrapper-interpreters the interpreter together with the specifications is the layer.

With this layered architecture, the following steps are needed to add a new search engine to ORWISE:

1. Design an ISE wrapper for the specific search engine by using a chosen wrapper toolkit. For example in the case of W4F this involves specifying the extraction rules in terms of the HEL extraction language from which a Java class is generated per each wrapped web source. By contrast, wrapper-interpreters are directly called from the ISE interface layer using the wrapper specifications as parameters.
2. Create types in the mediator database as subtypes of *SearchEngine* and *DocumentView*.
3. Design an ISE interface module as the overloaded Amos II foreign function, *orwise*, calling the ISE wrapper module from step 1.

Once step 1-3 are completed the ISE is already queryable directly through *orwise*. However, the queries can be complex and very ISE dependent. Efficient and transparent queries to an ISE therefore requires an additional step:

4. Design the rewrite rules needed for the ISE to translate between, e.g., *webSearch* calls and the particular *orwise* calls.

6 Translating ORWISE Queries

Queries calling the *webSearch* function combined with other Web document related predicates are translated to an equivalent but more efficient query containing optimized calls to the function *orwise* overloaded for specific ISEs. The function *webSearch* could be defined as a query calling *orwise* without any translation. However such untranslated execution may be significantly less efficient. In our example, the Google query is translated to the following *orwise* query:

```
SELECT given_name(e), family_name(e)
FROM Employee e, DocumentView d, Google gse
WHERE d = orwise(gse, given_name(e) + ' ' +
family_name(e), 20, 'www.csd.uu.se', 'english');
```

where the signature of *orwise* is Google specific. Here *orwise* for Google takes the parameters *query*, *result size*, *language restriction*, and *host*. The function is defined as a foreign AmosQL function that calls the underlying ISE wrapper for Google. The example illustrates the semantic rewrite of the original query by the translator, where several calls to *webSearch* and *host* are combined into one call to Google's *orwise*. The translator also added the default specifications of '*english'* as language and that only the first 20 results should be returned. The result of *orwise* is a stream of *GoogleView* objects. The translator for each ISE knows how to generate optimized *orwise* calls with specific parameters expressing ISE supported capabilities.

As shown in the example, queries to a search engine will contain subqueries expressed using the specific query language of the ISE, which is usually different for different ISEs. In the example above the string "`given_name(e) + ' ' + family_name(e)`" is an example of the construction of a conjunctive query to Google (it uses AND by default). During query translation, there are possible query transformations that can dramatically improve performance and result quality. We have implemented some translator rules to show the usefulness of the system and can utilize other results in related areas [5,6].

7 Summary

A flexible system for querying Internet search engines through an OO mediator database system was presented. The system has the following unique combination of features:

1. Data about both the search engine capabilities and the results they return were modeled in an OO *ISE schema* in a mediator database.
2. The ISE schema permits transparent queries to ISEs with different capabilities and result structures. The mediation facilities provide for processing heterogeneous queries that combine data from ISEs with data from other data sources.
3. New kinds of ISEs can be easily plugged in. The system assumes the ISEs are autonomous and outside the control of the query processor.
4. The system is designed to be independent of the wrapper toolkits used for specifying the ISE wrappers. Several such publicly available toolkits were evaluated to choose one for the implementation.
5. The query processor provides a mechanism to plug in OO search engine specific rewrite rules for translating OO queries into the parameterized *orwise* calls. The system is independent of the actual rewrite rules to utilize previous work in this area.

References

1. B. Adelberg: NoDoSe – A Tool for Semi-Automatically Extracting Structured and Semistructured Data from Text Documents, *SIGMOD 1998 Conference*: 283:294, 1998.
2. N. Ashish, C. Knoblock: Semi-automatic Wrapper Generation for Internet Information Sources. *CoopIS'97 Conference:* 160-169, 1997.
3. G. Arocena, A. Mendelzon: WebOQL: Restructuring Documents, Databases, and Webs. *In Proc. ICDE'98*, Orlando, 1998.
4. O. Bukhres, A. Elmagarmid (eds.): *Object-Oriented Multidatabase Systems.* Prentice Hall, 1996.
5. V. Christophides, S. Abiteboul, S. Cluet, M Scholl: From Structured Documents to Novel Query Facilities. *SIGMOD 1994 Conference*: 313-324, 1994.
6. S. Chaudhuri, U. Dayal, T. W. Yan: Join Queries with External Text Sources: Execution and Optimization Techniques. *SIGMOD 1995 Conference*: 410-422, 1995.
7. C. Chang, H. Garcia-Molina, A. Paepcke: Predicate rewriting for translating Boolean queries in a heterogeneous information system. *ACM Trans. on Information Systems*, 17(1), 1999.
8. Donald D. Chamberlin, Jonathan Robie, Daniela Florescu: Quilt: An XML Query Language for Heterogeneous Data Sources. *WebDB'2000*: 53-62, 2000.
9. A. Firat, S. Madnick, M. Siegel: The Caméléon Web Wrapper Engine, *First Workshop on Technologies for E-Services*, Cairo, 2000.
10. G. Fahl, T. Risch: Query Processing over Object Views of Relational Data*, The VLDB Journal* , 6(4), 261-281, 1997.
11. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, J. Widom: The TSIMMIS Approach to Mediation: Data Models and Languages. *Intelligent Information Systems (JIIS),* Kluwer, 8(2), 117-132, 1997

12. R. Goldman, J. McHugh, J. Widom: From Semisturctured Data to XML: Migrating the Lore Data Model and Query Language, *WebDB'99*, 1999.
13. J. Gruser, L. Raschid, M. Vidal, L. Bright: Wrapper Generation for Web Accessible Data Sources. *CoopIS'98 Conference*: 14-23, 1998.
14. R. Goldman, J. Widom: WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web. *SIGMOD 2000 Conference*: 285-296, 2000.
15. G. Huck, P. Fankhauser, K. Aberer, Erich J. Neuhold: Jedi: Extracting and Synthesizing Information from the Web. *CoopIS'98 Conference*: 32-43, 1998.
16. L. Haas, D. Kossmann, E. L. Wimmers, J. Yang: Optimizing Queries across Diverse Data Sources. *23rd Intl. Conf. on Very Large Databases (VLDB'97),* 276-285, 1997.
17. V. Josifovski, T. Risch: Integrating Heterogeneous Overlapping Databases through Object-Oriented Transformations, *25th Conference on Very Large Databases (VLDB'99),* 435-446, 1999.
18. T. Kistlera, H. Marais: WebL: a programming language for the Web. In *WWW7*, Brisbane, Australia, *http://www.research.digital.com/SRC/WebL/,* 1998.
19. D. Konopnicki, O. Shmueli. W3QS: A query system for the World Wide Web. *21st Conference on Very Large Databases (VLDB'95)*, 54–65, Zurich, Switzerland, 1995.
20. N. Kushmerick, D. Weld, R. Doorenbos: Wrapper Induction for Information Extraction. *IJCAI'97* Vol. 1: 729-737, 1997.
21. L. Liu, C. Pu: An Adaptive Object-Oriented Approach to Integration and Access of Heterogeneous Information Sources. *Distributed and Parallel Databases,* Kluwer, 5(2), 167-205, 1997.
22. L. Liu, C. Pu, W. Han: XWRAP: An XML-Enabled Wrapper Construction System for Web Information Sources. *ICDE 2000:* 611-621, 2000.
23. H. Lin, T. Risch, T. Katchanounov: Adaptive data mediation over XML data. To be published in special issue on "Web Information Systems Applications" of *Journal of Applied System Studies (JASS)*, Cambridge International Science Publishing, 2001.
24. G. Mecca, P. Merialdo, P. Atzeni: ARANEUS in the Era of XML. *IEEE Data Engineering Bulletin*, Special Issue on XML, September, 1999.
25. A. O. Mendelzon, G. Mihaila, T. Milo: Querying the World Wide Web. *International Journal on Digital Libraries*, 1(1), 54-67, April 1997.
26. A. Paepcke: An Object-Oriented View Onto Public, Heterogeneous Text Databases. *Proceedings of the Ninth International Conference on Data Engineering (ICDE'93),* 1993.
27. D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, J. Widom: Querying Semistructured Heterogeneous Information. In *Deductive and Object-Oriented Databases, Proceedings of the DOOD'95 conference*, 1995, LNCS Vol. 1013, 319-344, Springer 1995.
28. D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, J. Widom: Querying Semistructured Heterogeneous Information. In *Deductive and Object-Oriented Databases, Proceedings of the DOOD'95 conference*, 1995, LNCS Vol. 1013, 319-344, Springer 1995.

29. T. Risch, V. Josifovski: Distributed Data Integration by Object-Oriented Mediator Servers, To be published in *Concurrency – Practice and Experience J.,* John Wiley & Sons, *http://www.csd.uu.se/~udbl/publ/concur00.pdf*, 2001.
30. T. Risch, V. Josifovski, T. Katchaounov: Amos II Concepts, http://www.csd.uu.se/~udbl/amos/doc/amos_concepts.html, 2000.
31. A. Sahuguet, F. Azavant: Building Intelligent Web Applications Using Lightweight Wrappers, *Data and Knowledge Engineering*, 36(3), 283-316, March, 2001.
32. D. W. Shipman: The Functional Data Model and the Data Language DAPLEX, *TODS,* 6(1), 140-173, 1981.
33. A. Tomasic, L. Raschid, P. Valduriez: Scaling Access to Heterogeneous Data Sources with DISCO. *IEEE Transactions on Knowledge and Date Engineering*, 10(5), 808-823, 1998.
34. G. Wiederhold: Mediators in the architecture of future information systems, *IEEE Computer,* 25(3), 38–49, 1992.
35. XQuery: A Query Language for XML, W3C Working Draft, 15 February 2001, *http://www.w3.org/TR/xquery/*.

Warp-Edge Optimization in XPath

Haiyun He and Curtis Dyreson

School of EE and Computer Science
Washington State University, Pullman, WA 99164-2752
{hhe,cdyreson}@eecs.wsu.edu
http://eecs.wsu.edu/~cdyreson

Abstract. We describe the design and preliminary analysis of an optimization technique for XPath called *warp-edge optimization*. The XPath data model is a tree-like data model that has an edge from an element to each component in the content of that element. The edges are traversed in the evaluation of an XPath expression. A *warp edge* is an edge that is something other than a parent to child edge, i.e., an edge from an element to a sibling or to a grandchild. Warp edges can be dynamically generated and stored during query evaluation to improve the efficiency of future queries. We describe the implementation of warp-edge optimization as a layer on top of Xalan, the XPath evaluation engine from Apache. Experiments demonstrate that in the evaluation of some XPath expressions, the use of warp edges results in substantial savings of time.

1 Introduction

The explosive growth of the World-wide Web (web) has lead to an increase in the number of organizations that use the Extensible Markup Language (XML) to exchange data [1]. XML is a markup language for specifying the structure and semantics of text data and documents. XML avoids common pitfalls in language design, is extensible, platform-independent, and supports internationalization [2].

There are several query languages for XML data collections. Examples include Lorel [3], XQuery [4], XML-QL [5], and XSL Transformations (XSLT) [6]. An important component in many of these languages, especially those promulgated by the W3C, is XPath [7]. XPath is a language for addressing parts of an XML document. For instance the XPath expression `(//paragraph)[5]` locates the fifth paragraph element in a document. XPath expressions are a core component of all XSLT and XQuery programs.

In this paper, we propose an optimization technique for XPath called *warp-edge optimization*. The XPath data model is a tree-like data model that has an edge from an element to each component in the content of that element. The edges are traversed in the evaluation of an XPath expression. A *warp edge* is an edge that is something other than a parent to child edge, i.e., an edge from an element to a sibling or to a grand-

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 187-196, 2002.

child. Warp edges can be generated and stored during query evaluation to improve the efficiency of future queries. In the evaluation of some XPath expressions, the use of warp edges results in substantial savings of time since the warp edges connect nodes separated by two or more non-warp edges.

This paper makes several contributions. First, we discuss how to support warp-edge optimization by dynamically caching query results. Second, we implement the technique as a *layer* on top of, but separate from, an XPath evaluation engine. The important advantage offered by a layered architecture is that the warp-edge optimization layer can be combined with any XPath evaluation engine. Hence, we do not have to modify an XPath evaluation engine to optimize XPath queries. Third, we report on some experiments that demonstrate the efficacy of the technique.

The remainder of this paper is organized as follows. In the next section we motivate the technique. We then briefly sketch the implementation of the technique and experimental results.

2 Motivation

In this section, an example is provided to demonstrate warp-edge optimization. Consider a sample XML document shown in
Fig. 1. The fragment shows part of a novel, and is short for expository purposes. The document root is the `<doc>` element. Within the root are a title and a chapter. The chapter also has a title and has several sections. Each section has a title, a paragraph and a note.

```
<doc>
  <title>A Tale of Two Cities</title>
  <chapter>
    <title>The Journey</title>
    <section>
      <title>The Beginning</title>
      <para>It was the best of times…</para>
      <note>A famous opening line.</note>
    </section>
    <section>
      <title>The Middle</title>
      <para>The second city was
        <emph>squalid</emph> in a tepid way.
      </para>
      <note>This not quite so famous.</note>
    </section>
  </chapter>
</doc>
```

Fig. 1. A sample XML document

The XPath data model for the sample document is depicted in Fig. 2. The data model is constructed when the document is parsed. Details of the model extraneous to the example have been omitted (e.g., text nodes); only element nodes are shown. The id of the node is shown within the node.

Assume that a user of the digital library retrieves all of the sections by submitting the query `//section`. The warp edges created by the query are shown as dashed lines in Fig. 2. The warp edges connect the root with each section since the query starts at the document root and terminates at each section. The size of the sample document is small and the sections can be found quickly, but in general the document could contain thousands of sections.

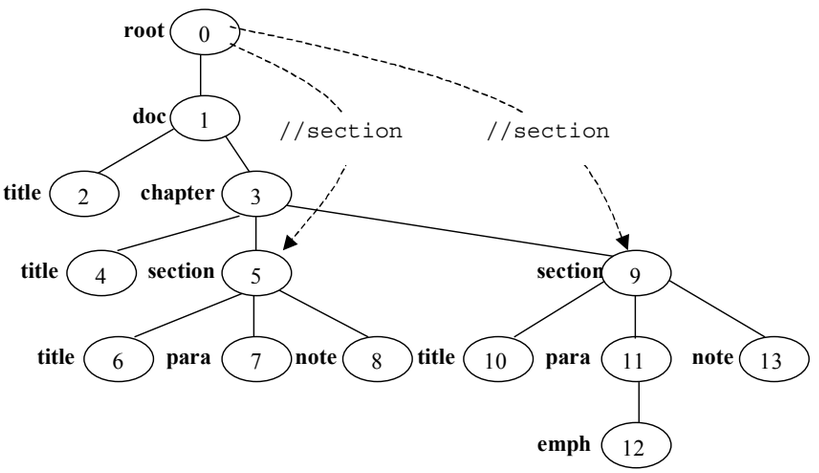


Fig. 2. The data model for the sample (warp edges are dashed lines)

XSLT and XQuery programs usually contain many XPath expressions. Assume that queries are subsequently given to retrieve the section titles, `//section/title`, and to retrieve the chapters that contain sections, `//chapter[section]`. Both queries can make use of the warp edges. The first query can traverse the warp edges to locate sections quickly, and then drop down to the title nodes. By using the warp edges the four edges on the path to each section node can be skipped. Not a big savings, but this document is quite small. Section 4 demonstrates the effectiveness of the technique on large trees. The second query can warp to sections, and then move up to find chapter nodes. This will be unlikely to result in a faster evaluation since four edges need to be traversed using both evaluation strategies (with and without warp edges). Sometimes following warp edges does not save time.

Related Work

There are two common technologies for query optimization in semistructured databases and XML query languages. The first is to build performance-enhancing data structures, e.g., indexes, and generate a query evaluation plan utilizing the structures. Lore has several indexes, such as value and path indexes [8]. Lorel queries can be

compiled into plans that make efficient use of the indexes [9]. Other path indexes include the t-index [10] and the Index Fabric [11]. For XPath, the Dynamic XML Engine (DXE) takes advantage of available indexes to accelerate queries [12]. Warp-edge optimization is similar because it builds a "path index" consisting of the warp edges. However, the index is constructed on-demand and in an ad hoc manner, unlike a DataGuide [13]. The above systems (except DXE) are database systems where the cost of statically building indexes is small in comparison to the benefits, whereas warp-edge optimization is applicable to in-memory XML parsing.

The second common technology is to rewrite the query (or batch of queries) to prune the search space. Gardarin, Gruser and Tang propose a technique to optimize linear path expressions and produce a cheap query execution plan [14]. Compile-time path expansion [15] utilizes a DataGuide (a schema) to prune the search space, while branching path optimization [16] recognizes that queries that follow the same branch in a tree can share the cost of exploring that branch, as does warp-edge optimization. Optimization in StruQL [17] combines both indexing and query rewriting. Warp-edge optimization dynamically implements branching path optimization.

3 Warp-Edge Optimization

A warp edge is an edge in the data model that traverses more than one level in the document tree. The canonical example is an edge to a grandchild node. Typically, warp edges are added as the result of previous queries. The warp edges can be traversed during the evaluation of a query

3.1 An Optimization Layer

There are two basic strategies for implementing warp-edge optimization. One approach is to modify the query evaluation engine to add warp edges to the data model. The second approach is to add a *layer* to perform warp-edge optimization above the legacy system. We adopt the layer approach because it is more flexible when the underlying system changes and can be implemented on proprietary evaluation engines. Any legacy system could be used such as Saxon [18], Sablotron [19], XT [20], Microsoft's XML Core Services (MSXML) 4.0, or Xalan [21]. Fig. 3 depicts the layered approach with the Xalan, the XPath evaluation engine from Apache. The warp-edge layer sits on top of the Xalan package. The layer takes a query and splits it into several sub-queries. Some of the sub-queries can be answered from the results of previous queries that are stored in the layer in an area called the *query cache*. The sub-queries that cannot be answered are sent to Xalan for evaluation. Xalan itself is not modified in any way, rather the layer uses Xalan's API. The layer processes the results of the sub-queries to build the result set. The results of sub-queries that generate new warp edges are stored in the query cache.

Caching the query results *induces* a set of warp-edges in the underlying data model. Whenever a new query result is added to the cache, in effect, it represents a corresponding warp edge in the parsed document. The opportunity to reuse query results increases as more results are added to the query cache.

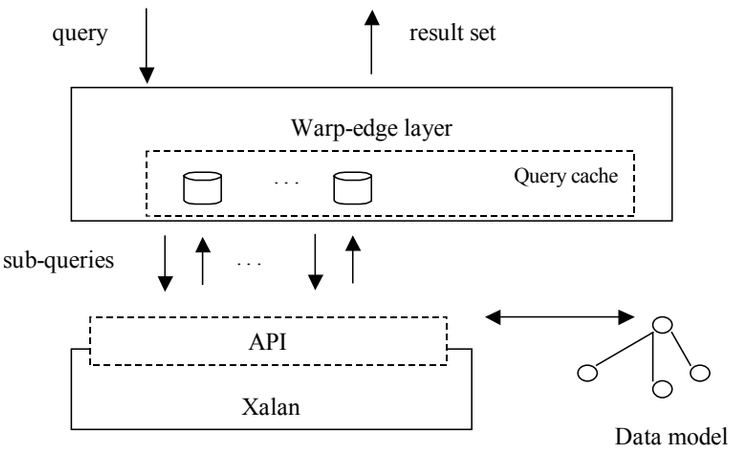


Fig. 3. Layer approach with Xalan

3.2 Prefix Matching

A query is evaluated by trying to find the *longest matching prefix* in the query cache. To facilitate the matching, the query cache is organized as a collection of trees called query cache trees (QCTs). When a query is issued, the layer first looks for the query in the cache. If the result is already there, the result is returned immediately because a new XPath query might be the same as an old one. On the other hand, a new XPath query can be different from all old queries, but we can still take advantage of the cached query results. The trick here is that we can use the cached result to "construct" the result for a new XPath query. For example, the new query may be an extension of an old query or may contain a part that has been evaluated before. The cached result is not returned immediately. Instead, it can be used as a temporary result to facilitate the evaluation of new query.

There are three outcomes to the prefix match.

1) *Full Cache Hit*: If the entire query is matched then the query is totally the same as a previous query and the result is already available.
2) *Partial Cache Hit*: A prefix of the query (i.e., the first few steps) match, then the cached prefix becomes the *context* for further evaluation of the remaining steps in the query. Prefix matching is performed for the rest of the steps in the query for each node in the context.
3) *Cache Miss*: This happens when a new query starts with a different step than all previous queries. A single step in the query is evaluated to establish a context for subsequent steps. Then the prefix match is tried against the remaining steps in the query.

At worst, the query is evaluated one step at a time by evaluating each step on the underlying query engine (every step results in a cache miss). Ideally, sequences of one or more steps can be found in the query cache (a cache hit). Then the cached result can be used without consulting the underlying query evaluation engine.

4 Empirical Analysis

In this section, we describe a preliminary set of experiments. Our goal is to test warp-edge optimization to determine whether it works under "ideal" conditions. The experiments involve tests on randomly generated data. We describe the parameters of each experiment in detail. Finally, we analyze the results.

4.1 Experimental Environment

We conducted the experiments on a Pentium PC (Dell Precision 340). It has an Intel® Pentium® 4 CPU 1700MHz, 512MB RAM and 37.2GB disk space. The PC runs Windows XP Professional Version 2002. We installed Java™ 2 v1.3.1_02 and Xalan-Java v2.3.1 for testing. The XML Parser used is Xerces-Java v2, which is available with the Xalan-Java package. We isolated the machine for testing. Only the test program and normal background processes are running during the testing period.

4.2 Random Experiment

We generated random XML documents for testing with the following configurable parameters.

- The children of root factor – This factor represents the number of children of the document root. It controls the top-level bushiness of the XPath data model tree.
- The depth factor – This factor represents the level of nesting of elements in the XML document. It controls the depth of the data model tree.
- The bushy factor - This factor describes the number of children in a non-leaf node in the data model tree. The bushiness can be fixed or chosen randomly from a range.

The tree is made random in two ways. First, the depth and bushiness of the tree can be made random to test with short, busy trees or deep, skinny trees, or some combination thereof. Because of limited memory, the trees are capped in size at approximately 12,000,000 nodes. Second, each level in the tree consists (almost) entirely of the same kind of elements, e.g., level one consists of `<A>` elements, level two of `<B>` elements, etc. However, we randomly convert up to 10% of the elements at each level into "magic" elements; a magic element is appended with a number e.g., `<B1>`. In a query, the magic elements can be used for node tests to limit the result-set size, e.g., the query "`//B/C`" will return far more nodes than "`//B1/C1`".

4.2.1 High Match Probability Experiment

In this experiment, we tested the performance using XPath queries that have high match probabilities, i.e., there is a greater chance to retrieve a large result set. We tested the following query batches on the randomly generated XML documents. The query cache is updated after each batch.

```
Batch 1:      /descendant-or-self::C
              /descendant-or-self::E

Batch 2:      /descendant-or-self::B/child::C
              /descendant-or-self::D/child::E

Batch 3:      /descendant-or-self::C/child::D
              /descendant-or-self::C/descendant-or-self::E
              /descendant-or-self::C/descendant-or-self::F
```

The above query batches only include "pure elements" and therefore have large result-sets. This simulates the situation where a user requests popular information from an XML document. Furthermore, the batches are designed to favor warp-edge optimization since the last two query batches utilize the warp edges. We tested a range of root children, depth factors and bushy factors independently, and averaged the results of the tests on runs of five random trees.

In the first experiment, we varied the number of root children and fixed the depth factor to be 6 and bushy factor to be 3. The turnaround time result is shown in the left-hand graph in Fig. 4 (including the time to update the QCT). The right-hand graph shows the space overhead of the QCT.

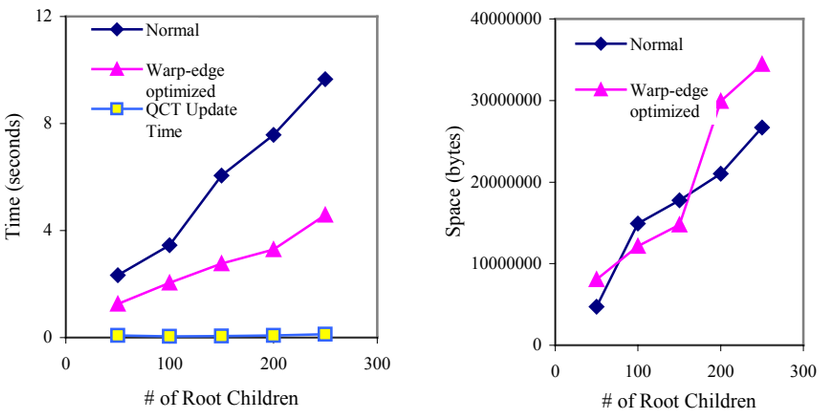


Fig. 4. Varying the number of root children

Next, we varied the depth factor but fixed the number of root children to be 50 and the bushy factor to be 3. Then the XML document is moderately bushy with a moderate number of sub-trees, but varies from shallow to deep. We obtain the graphs in Fig. 5.

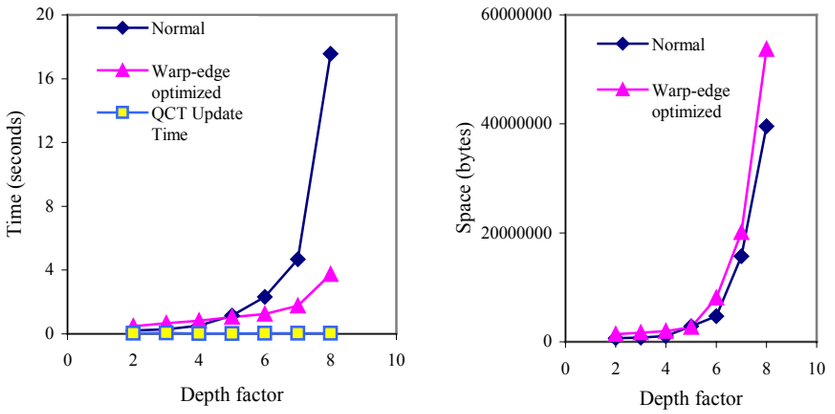


Fig. 5. Varying the depth factor

Third, we varied the bushy factor and fixed the other parameters, i.e. the number of root children is 50 and the depth factor is 5. Then the XML document will have a moderate number of sub-trees and be of moderate depth, but will vary in bushiness from skinny to fat trees. By this means, we can see how our approach performs with a change in bushiness. The results are shown in Fig. 6.

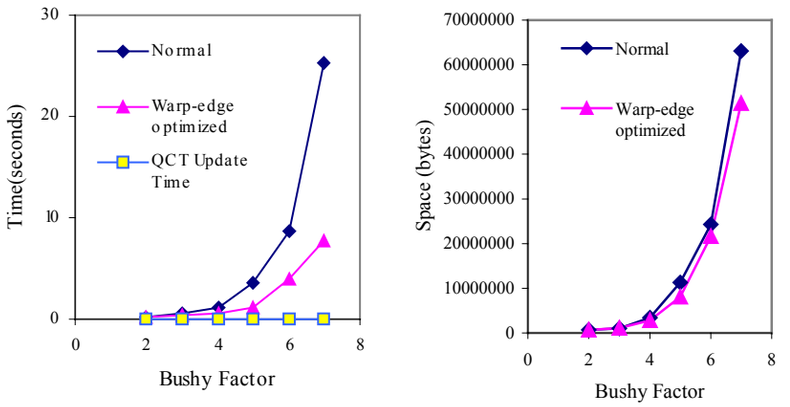


Fig. 6. Varying the bushy factor

The turnaround time for query evaluation shows that the optimization is working best for deeper and bushier trees. In the first experiment, the optimization approximately halves the time needed for query evaluation at a modest increase in the amount of space. In the second experiment, although query performance degrades exponential (as the size of the tree increases exponentially), the non-optimized query time in-

creases much faster than that of the optimized query. For very deep trees, when the depth reaches 8, the optimization provides a five-fold increase in query performance. Again, only a small amount of additional space is needed for the optimization. The third experiment, testing trees of varying bushiness, confirms that the optimization can improve query performance when more warp edges are utilized in the larger and bushier trees.

The graphs also depict the time needed to update the query cache trees (QCT). The update time is not counted in the turnaround time. The cost however, is usually quite trivial. The reason is that the QCT update just generates a mapping between the query and the corresponding result set, which is not a time-consuming process.

Overall, the experiments show that while warp-edge optimization needs a small amount of additional space, it can improve query performance for large, deep, and bushy trees.

5 Conclusions and Future Work

In this paper we described the design and analysis of an optimization technique for XPath called warp-edge optimization. Warp edges can be dynamically generated and stored during query evaluation to improve the efficiency of future queries. We implemented warp-edge optimization as a layer on top of Xalan, the XPath evaluation engine from Apache. Experiments demonstrate that in the evaluation of some XPath expressions, the use of warp edges results in substantial savings of time at a modest increase in space. The benefit of the layered implementation is that warp-edge optimization can be wrapped around any back-end XPath evaluation engine. Our experiments show that the cost of the layer is small.

In future, we plan to develop query rewrite rules to support more effective use of the cache in a manner similar to rewriting database queries using materialized views. Also, since the cache independently maintains some information, we believe that query caching can be used to provide partial answers when the original document is no longer available or expensive to query directly.

References

1. World Wide Web Consortium. Extensible Markup Language (XML). http://www.w3c.org/XML. Current as of October 2000.
2. World Wide Web Consortium. XML in 10 points. http://www.w3c.org/XML/1999/XML-in-10-points. Current as of November 2001.
3. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data**.** SIGMOD Record, 26(3):54-66, September 1997.
4. World Wide Web Consortium. XQuery 1.0: An XML Query Language. http://www.w3c.org/TR/xquery/. Current as of April 2002.

5. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. WWW10, Toronto, CA.
6. World Wide Web Consortium. XSL Transformations (XSLT) Version 1.0. http://www.w3c.org/TR/1999/REC-xslt-19991116. Current as of November 1999.
7. World Wide Web Consortium. XML Path Language (XPath) Version 1.0. http://www.w3c.org/TR/xpath. Current as of April 2002.
8. J. McHugh and J. Widom. Query Optimization for XML. In Proceedings of VLDB, Edinburgh, Scotland, September 1999.
9. J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing Semistructured Data. Technical Report, Stanford University, Database Group, January 1998.
10. T. Milo and D. Suciu. Index structures for path expressions. In ICDT'99, Jerusalem, Israel, January 10-12, 1999, pages 277-295, 1999.
11. B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. In Proceedings of VLDB, September 2001, pp. 341-350.
12. Exceloncopr. Optimizing XPath Expressions. http://support.exceloncorp.com. Current as of May 2001.
13. R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In Proceedings of VLDB, August 1997, pp. 436-445.
14. G. Gardarin, J.Gruser, and Z. Tang. Cost-based Selection of Path Expression Processing Algorithms in Object-oriented Databases. In Proceedings of VLDB, Bombay, India, pp. 390-401.
15. J.McHugh and J. Widom. Compile-Time Path Expansion in Lore. In Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, Jerusalem, Israel, January 1999.
16. J. McHugh and J.Widom. Optimizing Branching Path Expressions. Technical report, Stanford University, Database Group, June 1999.
17. M. Fernandez, D. Florescu, J.Kang, A. Levy, and D. Suciu. Catching the Boat with Strudel: Experiences with a Web-site Management System. In Proceedings of SIGMOD, Seattle, Washington, June 1998, pp. 414-425.
18. Michael Kay. SAXON The XSLT Processor. http://saxon.sourceforge.net. Current as of February 2002.
19. Ginger Alliance. Sablotron XSLT, DOM and XPath processor. http://www.gingerall.com/charlie/ga/xml/p_sab.xml. Current as of March 2002.
20. James Clark. XT. http://www.jclark.com/xml/xt.html. Current as of November 1999.
21. Apache XML Project. Xalan-Java version 2.3.1. http://xml.apache.org/xalan-j/index.html. Current as of March 2002.

A Caching System for Web Content Generated from XML Sources Using XSLT

Volker Turau

FH Wiesbaden, University of Applied Sciences, Department of Computer Science
Kurt-Schumacher-Ring 18, 65197 Wiesbaden, Germany
{turau}@informatik.fh-wiesbaden.de
http://www.informatik.fh-wiesbaden.de/~turau/

Abstract. A large number of web sites is database-driven: content stored in databases is formatted at request time into HTML pages. Increasingly, the data is first sampled into XML-documents and then transformed into their final representation format using XSLT. This process has a lot of advantages, but it is very resource-intensive and is currently not suitable for servers with high traffic. This paper presents a novel caching system for this scenario to reduce the server load and to increase scalability.

1 Introduction

An important aspect of the development of the World Wide Web during the last years is the increasing use of dynamically generated documents. The documents are created at request time using techniques such as CGI, ASP, PHP, and JSP. This allows the personalization of documents and the inclusion of external content resources. Requirements such as multi-publishing (HTML, WML, and PDF) and internationalization, as well as the need for frequent layout changes have led to a strong decoupling of content and presentation. A new paradigm and new standards have emerged. Content is described using XML and presentational views are constructed using XSLT-transformations. More and more content is stored directly in an XML-format in content management systems. But a large amount of data is still not in XML-format. This will probably change in future, but there will always exist different formats for dedicated applications. A large variety of tools is available for converting data into XML-formats, e.g. for relational databases, directory and naming services or semistructured text.

A common scenario is to perform the conversion into XML and the transformation into HTML based on user input and cookie values at request time. The main problem with this approach is that it does not scale when the number of concurrent users grows. The reasons are the high database load and the high consumption of resources during the transformation phase: XSLT-transformations are CPU-intensive and main-memory representations of XML-documents using the document object model are memory-intensive. This can seriously reduce Web server performance. According to [5] it is not uncommon for a program to consume over a second of CPU time in order to generate a single dynamic page.

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 197–207, 2002.

Caching is currently the primary mechanism in the WWW for reducing the latency as well as bandwidth requirements. Numerous tools and techniques have been proposed and successfully used for caching static content. If successful, caching can provide significant additional benefit by reducing server load and end-to-end latency. For obvious reasons, caching has been primarily applied to static content. Just recently research on caching dynamic Web content has begun.

In this paper we present a novel server-based caching system to improve the performance and scalability of the above described scenario using XML and XSLT. The caching system is based on the following observations:

- Most applications are read-heavy, a significant percentage are re-reads,
- Many forms of personalizations can be achieved with parameterized XSLT-style-sheets operating on several documents, hence XML-documents can be reused in more transformations,
- XSLT-style-sheets change rarely, and
- XML-documents change more often, but still are valid for a limited time.

The main ideas of our approach are: configurable caching at several levels, caching of the documents in a form suitable for fast processing (i.e. objects instead of a textual format) and spooling of purged objects on persistent storage. The caching system is completely transparent for content authors and designers.

2 Caching Techniques

Caching is performed in various locations throughout the web: Web Servers, proxies and Web clients (for an introduction see [12]). Not every document on the web is cacheable. Dynamically generated documents are typically considered uncacheable. Examples of such documents are fast-changing content like stock quotes, personalized pages, and search engine query results. One way to allow for the caching of dynamic content is to cache programs that generate or modify the content [4]. Another approach is to enable servers to cache portions of documents to optimize server-side operations. Several server extensions exist for this purpose [6]. Another proposal is to isolate the static parts of documents in order to keep them in caches. The final documents are then reassembled at the client [2, 10]. Sending the differences between pages or between versions of a page has been proposed in [9]. Caching frameworks for database-backed web sites have received a lot of attention recently ([3, 5, 13]). These studies consider passive caching of query results, XML fragments and HTML pages generated from DBMS-resident data. In contrast [8] considers active caching exploiting the query semantics of HTML forms. Most proposals for caching dynamic content are not transparent for authors, e.g. special notation marks cacheable fragments. This is contrary to our approach, where caching and authoring are orthogonal concepts.

3 Scenario

Fig. 1 depicts a scenario for generating web pages, which can be found on many database-driven web sites today. Special tools generate XML-documents from data stored in databases or legacy applications. The extraction-process is described by dedicated extraction-languages, which are often extensions of SQL. Examples of such systems are XML-SQL Utility from Oracle, XML Extender for DB2 from IBM, Silkroute or DB2XML [7, 11]. Using XSLT-style-sheets these XML-documents are then transformed into their final presentation format (e.g. HTML, WML or PDF). In this simple form, the complete process from the database to the document in its final presentation form is very resource intensive. The main bottlenecks are: querying the data source (e.g. a database), generating the XML-documents, building the internal representation of XSLT-style-sheets, and performing the transformation.

A crucial observation is that many of the external parameters used for the extraction process can be shifted into the transformation style sheets. This may lead to longer XML-documents, at the other hand these documents can be reused in different requests. This makes the documents ideal candidates for caching.

In this paper we propose a caching system to improve scalability, and to reduce user perceived latency in the above described scenario. The performance gain is achieved by reducing the number of trips to the database or other external sources of information, avoiding the cost of repeatedly recreating objects, and sharing objects between threads. In a database-driven web site the caching system dramatically reduces the database load, since the number of database accesses drops considerably.

4 The Caching System

At the core of the caching system is a servlet which handles all incoming requests. It maps all requests with a given prefix to a corresponding transformation document (based on the name of the requested document). This specifies the XML-document and the available XSLT-style-sheets. We have developed a

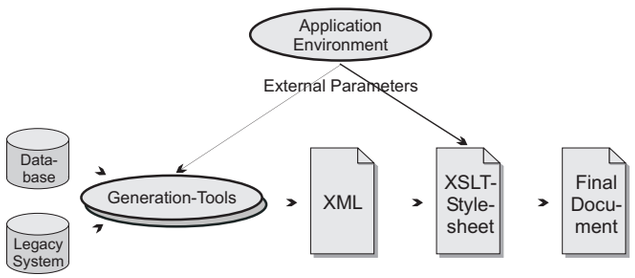


Fig. 1. Scenario for database-driven web sites

document type definition describing the transformation process. Fig. 2 shows the components used by the caching servlet. The three caches store different kinds of entities. The *Document-Cache* keeps individualized documents in their final presentation format. The other two caches store Java objects representing documents and style sheets. The *XML-Cache* keeps XML-documents in a format suitable for fast XSLT-transformation and the *XSLT-Cache* stores objects which perform the XSLT-transformation.

All objects within the three caches are shared. However, access to cached objects is not serialized by access locks, allowing for a high level of concurrent access. The transformation documents are stored in the file system. Upon receiving a request for a document, the servlet performs the following steps:

1. From the incoming request, the servlet determines the corresponding transformation document.
2. The servlet extracts parameters from cookies and the HTTP-request and determines the MIME type for the document (e.g. from the user-agent header).
3. From this data a document descriptor object *ddo* is constructed (including a key for the final document based on a hash value of the *ddo* including parameter values).
4. If the document is cacheable and is contained in the Document-Cache and is not stale it is sent to the requesting client. HTTP headers such as `If-Match` and `If-Modified-Since` are recognized and treated accordingly. Otherwise the following steps are performed:
   (a) If the XML-document is cacheable it is fetched from the XML-cache (if it is not present or stale the object representing the document is read from disk or generated and the cache is updated) otherwise it is generated,
   (b) If the style sheet is cacheable the XSLT-transformer is fetched from the XSLT-cache (if it is not present or stale the object representing the style sheet is read from disk or generated and the cache is updated) otherwise the style sheet is read from the file system, parsed and a transformer object is generated,
   (c) Parameters are mapped to style sheet variables, the transformation is applied, the Document-Cache is updated, the `Cache-Control` header

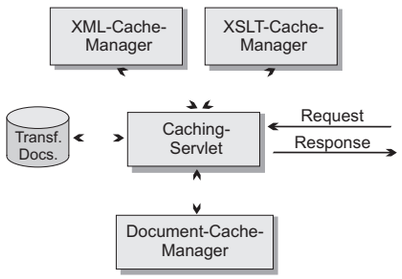


Fig. 2. Overall architecture

for the HTTP response is set (enabling network or client caches to store the documents) and the final document is sent to the requesting client.

The individual steps of this process are controlled by the content of a transformation document. These documents are written by the a person who can estimate expiration dates, access frequencies, documents sizes etc. Content authors and designers need not be aware of the existence of these documents. The CachingServlet stores the transformation documents in an in-memory data-structure for faster access. The XML-Cache-Manager maintains an index with information on how to construct the requested documents from underlying data sources (e.g. databases).

4.1 The *Transformation* DTD

Fig. 3 shows the main elements and attributes of the *transformation* DTD. The parameters for the usage of the three caches are described at the corresponding elements using the attributes `useCache` and `checkIfCacheStale`. The first attribute indicates whether the cache should be used at all. The cache cannot be used if the transformation process has side effects (e.g. updating persistently stored data). The second attribute determines, when a cache entry should be checked for staleness. Possible values for this attribute are `always, never` and `ifOlderThan=nnnn`, where `nnnn` denotes a time span in seconds. Note the default values for these attributes are `true` and `always`.

The subelement `xml` refers to the XML-document to be transformed. The attribute `name` references an entry in an index of the XML-Cache. The index contains information on how to construct the document from underlying data sources. The subelement `xslt` provides information relevant for the transformation process. With each `stylesheet` element a XSLT-style-sheet producing a different MIME type is associated. The decision which style sheet to use is made at run time by the servlet based on the request details. The `parameter` element allows the specification of a mapping from names for request parameters and cookies to style sheet parameters. Furthermore, a default value for each parameter must be given. The handling of the parameters is independent of the MIME

```
<!ENTITY % common.att 'name CDATA #REQUIRED useCache (true | false) "true"
                       checkIfCacheStale CDATA "always"'>
<!ELEMENT transformation (xml, xslt*)>
<!ATTLIST transformation %common.att;>
<!ELEMENT xml EMPTY>
<!ATTLIST xml %common.att; storePersistent (true | false) "true">
<!ELEMENT xslt (parameters?, stylesheet+)>
<!ELEMENT parameters (parameter)+>
<!ELEMENT parameter EMPTY>
<!ATTLIST parameter name CDATA  #REQUIRED externalName CDATA #IMPLIED
                    defaultValue CDATA #REQUIRED>
<!ELEMENT stylesheet EMPTY>
<!ATTLIST stylesheet %common.att; storePersistent (true | false) "true"
   maxTransformers CDATA "3" mimetype CDATA #IMPLIED>
```

Fig. 3. The transformation DTD

type of the final document. Therefore, `parameter` is not a subelement of the `stylesheet` element, but is provided at a higher level.

Fig. 4 depicts an example document according to the `transformation` DTD. The logical name of the document is `bookOrder`. The document may be placed into the document cache, but staleness must be checked before every access. The XML-document is considered to be stale if it is older than 2 hours. There are two style sheets for this document: one for pure web clients and one for WAP-clients. The Caching-Servlet does never check whether the two cached style sheets are stale. They only will be replaced if the XSLT-Cache-Manager gets notified by an external application. Of course, at any time the style sheet can be discarded from the cache as a consequence of the cache replacement strategy. The style sheet has two parameters `Name` and `Value`. If the request contains a parameter or a cookie with name `Name`, this value will be forwarded to the style sheet. If the request specifies a parameter with name `v`, this will be mapped to the style sheet parameter `Value`, otherwise the parameter `Value` will be 10.

4.2 The XML-Cache

The XML-Cache-Manager administrates a cache of XML-documents represented in a format optimized for the XSLT-processor. This relieves the processor to rebuild this data structure for every request. These documents are never updated, hence a read-only data structure is sufficient (in contrast to updatable formats such as DOM). Note that the XML-Cache is kept in main memory. Fig. 5 depicts the main components of the XML-Cache. The XML-index is a data structure storing information about the XML-documents: name, references to objects in the cache and on disk, generation and expiration date etc. Furthermore, information to generate the documents is also available (a reference to an external application and configuration parameters). XML-documents can be generated by different applications (see section 3) from data contained in databases or other resources. All access to the cache is performed exclusively by the XML-Cache-Manager. This component keeps the XML-index in a consistent state.

The generation tools can be triggered by changes in the databases through a trigger component. Alternatively they can be started by the XML-Cache-

```
<!DOCTYPE transformation SYSTEM "transformation.dtd">
<transformation name="bookOrder">
  <xml name="order" checkIfCacheStale="ifOlderThan=7200" storePersistent="false"/>
  <xslt>
    <parameters>
      <parameter name="Name" defaultValue="unknown"/>
      <parameter name="Value" externalName="v" defaultValue="10"/>
    </parameters>
    <stylesheet name="orderStandardWeb" maxTransformers="5"
                checkIfCacheStale="never" mimetype="text/html"/>
    <stylesheet name="orderStandardWap" maxTransformers="2"
                checkIfCacheStale="never" mimetype="text/vnd.wap.wml"/>
  </xslt>
</transformation>
```

Fig. 4. An example of a transformation document

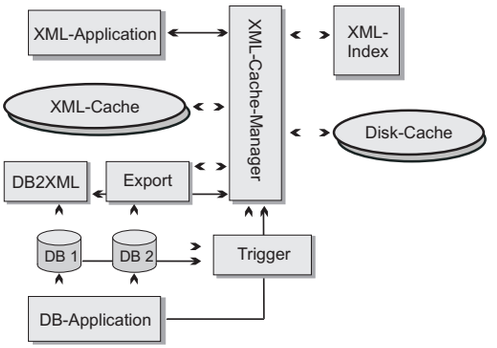


Fig. 5. The XML-Cache

Manager or by external applications. A key problem is to determine which documents are affected by changes of the underlying data. For example, a set of cached documents may be constructed from different tables or even tables from different databases. A method is needed to determine the necessary updates. To avoid unnecessary rebuilding of cached documents, the method should describe the dependencies between the external content and the documents in as precise a fashion as possible.

For documents generated from databases the XML-Cache-Manager keeps a list of names of database tables. If the manager is informed by a DB-Trigger or an external application that the data in a table has changed, the affected documents are marked as invalid in the cache. This technique is similar (albeit simpler) to the *Data Update Propagation* algorithm presented in [5]. The main difference is the granularity of the cached objects. While we cache only complete XML-documents, in [5] fragments of HTML documents are stored in the cache.

A low priority thread periodically checks all entries for staleness and marks them as invalid. When an object is invalidated, the invalid version of the object will remain in the cache as long as it is used in a transformation. It is thus possible to have multiple versions of an object in the cache at the same time, however, there is never more than one valid version of the object. The old or invalid versions are only visible to transformations that where started before it was invalidated. If the external data of a document is updated a new copy of the object is created in the cache and the old version is marked as invalid. This allows access to objects without requiring any read locks. The strategy is similar to that proposed in [1].

Valid objects remain in the cache as long as there is space available. When the cache capacity is reached, valid objects will be discarded from the cache based on their usage pattern (currently only *least recently used* is implemented). Since the objects in the cache are basically instances of Java classes, they can be serialized and spooled to a disk cache rather than being destroyed. This option is controlled by the attribute `storePersistent` of the element `xml`. If the document is later

requested again, the object is loaded from disk and deserialized. Objects in the disk cache may be removed as well to meet space restrictions, however, the disk cache is presumably much larger than the memory cache so objects will be less likely to be purged. Invalid objects are removed from both the memory and disk cache. The disk cache is also used, when the server is temporarily shutdown and started again.

The Cache-Manager has an interface to plug-in modules for logging and monitoring. This interface also supports dynamic changes of configuration parameters.

4.3 The XSLT-Cache

Before XSLT-style-sheets can be processed, they are converted into some internal machine-readable format. This step is performed using a XML-Parser. Since style sheets rarely change, it is advantages to keep the machine-readable representation in memory for repeated use. This process is called style sheet compilation. Different XSLT-processors implement style sheet compilation differently. The caching servlet requests a compiled style sheet through the XSLT-Cache-Manager.

Compiled style sheets become stale if the underlying style sheets change. Therefore, the XSLT-Cache-Manager checks the date the file containing the style sheet was last modified. The details of the cache administration are similar to those of the XML-cache (including the use of a disk cache and the avoidance of read locks). Before a compiled style sheet is actually used in a transformation process, the parameters provided by the request (including values of cookies) are mapped to the style sheet parameters as indicated in the transformation document. The structure of the XSLT-index is similar to that of the XML-index. Fig. 6 depicts the main components of the XSLT-Cache.

4.4 Document-Cache

The Document-Cache operates like most other server-side web caches. It keeps copies of the generated documents together with some meta data (date of last modification, expiration date etc.). This information is inserted into the corresponding HTTP-header, hence intermediate proxies can place this document in

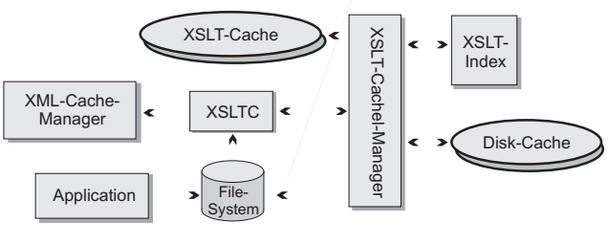


Fig. 6. The XSLT-Cache

their caches. Only documents which are tagged as cacheable in the corresponding transformation document are inserted into the Document-Cache. Note that documents in the cache are indexed using the value of the attribute `name` of the element `transform`, the style sheet name, and the values of the parameters forwarded to the style sheet.

5 The Implementation

The caching system is implemented as a servlet using the Java Servlet API version 2.3. The servlet is hosted by the Servlet-Container Tomcat version 4.0, which is accessed via the Apache Web Server. The handling of XML-documents and the XSLT-transformations are based on the tools provided by the Apache XML project: Xerces for XML parsing and Xalan for XSLT-processing. The servlet uses JAXP version 1.1 to access Xerces and Xalan. XSLT-style-sheets are compiled into so called *Translets* using the XSLTC API. These are Java classes which are executed at runtime to perform the translation. Note that the style sheet compiler XSLTC has access to the XML-Cache. In case a XSLT-style-sheet references another XML-document using the *document()* function of XSLT, the compiler can load such documents from the cache.

Thread safety is an important issue in web applications, where many requests share the same style sheet. To perform repeated transformations, XSLTC provides so called *Transformer* objects. These are lightweight objects that track state information during the transformation and parameter values specific to the current request. For this reason, a separate transformer instance must be used for each concurrent transformation. Therefore, transformer objects cannot be kept in a cache similar to XML-documents. To reuse a transformer, the cache maintains for each style sheet a pool, which keeps several instances of transformers. The size of the pool is controlled by the value of the attribute `maxTransformers`. The caching servlet requests such an object through the XSLT-Cache-Manager, if none is available, the request is blocked until a suitable object is returned to the cache. After the transformation process the transformer object is returned to the pool and can be used again. The number of instances of transformer objects per style sheet limits the number of concurrent users of this style sheet.

Throughout the implementation a high level goal was to minimize synchronization bottlenecks. For this reason Java core classes such as `Hashtable` were barely used. In our test-implementation the XML-documents were generated with the DB2XML tool and with the Xalan SQL library from data stored in relational databases.

6 Conclusion and Future Work

In this paper we have presented a caching system for database-driven web sites using XML and XSLT. The primary goal was to increase scalability and to reduce the load of the backend systems storing the data. This is achieved by caching and spooling XML-documents and style sheets as Java objects as opposed to

a plain textual format. Since each object carries its own expiration time, the system is very flexible and integrates well with HTTP. The caching mechanism is completely transparent for authors and designers.

A preliminary performance analysis revealed, that in many configurations the caching system reduces the server load and lessens user-perceived delays (compared to no caching). Higher performance gains were achieved for larger XML-documents and larger XSLT-style-sheets (because the high initialization time of these objects is saved). The same observation was made for the spooling of Java objects to disk. In this case the number of concurrent users could almost be doubled compared to the non-caching approach, while achieving the same response time. The benefits are higher when XML documents can be used in more request due to parameterized style sheets (because less cache memory is used).

Currently we are building an extensive testbed to evaluate the benefit of the caching system. This proves to be a difficult task. This is due to the large number of factors that influence the result: size of XML documents and style sheets, volume of database queries, cache replacement strategies, pool configuration parameters, access patterns, available main memory etc. Furthermore, we investigate possibilities to reflect changes of the data in the database by updating the objects in the XML-Cache rather than building them from scratch.

References

[1] Bortvedt, J., Functional Spec. for Object Caching Service for Java, Oracle, 2002. 203

[2] Braband, C., Møller, A. Olesen, and Schwartzbach, M. I., Language-Based Caching of Dynamically Generated HTML, to appear in World Wide Web Journal. 198

[3] Candan, K. S., Li., W.-S., Luo, Q., Hsiung, W.-P., and Agrawal, D. Enabling Dynamic Content Caching for Database-Driven Web Sites, SIGMOD Conf. 2001. 198

[4] Cao, P., Zhang, J. and Beach, K., Active Cache: Caching dynamic contents on the Web, Proc. Middleware conference, 1998. 198

[5] Challenger, J., Iyengar, A., and Dantzig, P., A Scalable System for Consistently Caching Dynamic Web Data, Proc. IEEE INFOCOM, 1999. 197, 198, 203

[6] Davison, B. D., A Web Caching Primer, IEEE Internet Comp., Nr. 4, 38 - 45, 2001. 198

[7] Fernández, M., Morishima, A., and Suciu, D., Publishing Relational Data in XML: the SilkRoute Approach, Proc. ACM SIGMOD Int'l Conf. Manag. of Data, 2001. 199

[8] Luo, Q., and Naughton J. F., Form-Based Proxy Caching for Database-Backed Web Sites, Proceedings of the 27th VLDB Conference, Roma, Italy, 2001. 198

[9] Mogul, J. et al., Potential Benefits of Delta-Encoding and Data Compression for HTTP, Proc. ACM SIGCOMM, 181 - 194, 1997. 198

[10] Nottingham, M. et. al., ESI Language Specification 1.0, W3C Note, August 2001. 198

[11] Turau, V., Making legacy data accessible for XML applications, Technical Report, University of Applied Sciences, FH Wiesbaden, 1999. 199

[12] Wessels, D., Web Caching, O'Reilley & Associates, Sebastopol, Cal. 2001. 198
[13] Yagoub, K., Florescu, D., Issarny, V., and Valduriez, P., Caching Strategies for Data-Intensive Web Sites, Proc. VLDB 2000. 198

Finding Similar Queries to Satisfy Searches Based on Query Traces

Osmar R. Zaïane and Alexander Strilets

University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{zaiane,astrilet}@cs.ualberta.ca

Abstract. Many agree that the relevancy of current search engine results needs significant improvement. On the other hand, it is also true that finding the appropriate query for the best search engine result is not a trivial task. Often, users try different queries until they are satisfied with the results. This paper presents a method for building a system for automatically suggesting similar queries when results for a query are not satisfactory. Assuming that every search query can be expressed differently and that other users with similar information needs could have already expressed it better, the system makes use of collaborative knowledge from different search engine users to recommend new ways of expressing the same information need. The approach is based on the notion of quasi-similarity between queries since full similarity with an unsatisfactory query would lead to disappointment. We present a model for search engine queries and a variety of quasi-similarity measures to retrieve relevant queries.

1 Introduction

Many commercial search engines boast the fact that they have already indexed hundreds of millions of web pages. While this achievement is surely remarkable, the large indexes without doubt compromise the precision of the search engines adding to the frustration of the common users. Many agree that the relevancy of current search engine results needs significant improvement. Search engines present users with an ordered and very long list of websites that are presumably relevant to the query specified based on criteria specific to each different search engine. Users typically consult the first ten, twenty or maybe thirty results returned and give up if relevant documents are not found among them. Results are normally ranked by relevance, which is calculated based mainly on the terms present in the query and not necessarily on the semantics or meaning of the query. Some Search engines like Google [2] use the notion of incoming and outgoing hyperlinks from documents containing the query terms to rank the relevant URLs [1]. It remains, however, that these lists are too long to browse. While users are at the mercy of the result ranking procedure of the search engine, they are

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 207-216, 2002.

also constrained by the expressiveness of the query interface, and often have difficulty articulating the precise query that could lead to satisfactory results. It happens also that users may not exactly know what they are searching for and thus don't know how to effectively express it and end up selecting terms for the search engine query with a trial and error process. Indeed, often users try different queries until they are satisfied with the results. If the results are not satisfactory, they modify the query string and repeat the search process again. Many commercial search engines provide possibilities to narrow searches by either searching within search results or augmenting a query to help users narrow their search [8]. However, these query augmentations just append terms to the existing terms in the query. For example a search on AltaVista with the term "avocado" leads to the following suggestions: avocado trees, growing avocados, avocado recipes, avocado oil, avocado plant, etc. This is comparable to our first quasi-similarity measure presented below. We call this method the "Naïve approach" because it simply looks in the query trace for queries having terms similar to the terms in the current query. In other words, it considers only the terms in the queries and simply performs intersections (see below). This example, however, shows that it is conceivable that more than one user would send a search query for a similar need and it is possible that these queries are differently expressed. A study by Markatos shows that many queries sent to a search engine can be answered directly from cache because queries are often repeated (by presumably different users). The study reports that 20 to 30% of queries in an Excite query trace with 930 thousand queries, were repeated queries [6]. A similar study using AltaVista query logs demonstrated that each query was submitted on average four times. This average is for identical queries not taking into account upper/lower case, word permutations, etc. This justifies the assumption that when one user submits a query to a search engine, it is highly likely that another user already submitted a very similar query. The queries can be identical as found and reported by the studies mentioned above, or articulated differently with different terms but for the same information needs. This is the major argument to put forward the idea of using query collective memory to assist individual users in articulating their information needs differently by suggesting quasi-similar queries. We define quasi-similarity in the next section.

The idea of tapping into the collective knowledge of users, embodied as a set of search queries, is not new. Fitzpatrick et al. studied the effect of using past queries to improve automatic query expansions in the TREC (Text REtrieval Conference) environment [4]. The idea is that top documents returned by the query from a pool of documents are also top documents returned by similar queries and are good source for automatic query expansions. They compared the performance of this method against the unexpanded baseline queries and against the baseline queries expanded with top-document feedback. The authors present a query similarity metric that empirically derives a probability of relevance. They also introduce the notion of threshold to control on per query basis whether or not a query should be expanded. In a similar study, Glance describes the community search assistant, a software agent that recommends queries similar to the user query [5]. The similarity between queries is measured using the number of common URLs returned from submitting both queries to the search engine. The main contribution is the notion of collaborative search using query traces in the web search engine context. However, if two queries have the same search

results and the user is not satisfied with the result of one of them, the results of the second ought to be unsatisfactory. Thus, the suggested similar query is inadequate; hence, the notion of quasi-similarity of queries presented in the next section.
A query trace is basically a log containing previously submitted queries. This log is not enough to compute similarities between queries. In Section 2 we introduce Query Memory, a data structure that holds not only the collective query trace but also extra information pertaining to the queries that would help in measuring similarities between queries. We introduce our query quasi-similarity measures using the Query Memory in Section 3. In Section 4 we depict our prototypical implementation. Some examples are discussed in Section 5. Finally, Section 6 presents our conclusions.

2 Query Memory Data Structure

We have collected a large Query Memory from a popular meta-crawler and saved these queries locally in our database. A query in our view is not just a string, but a bag of words and associated to it is the list of documents that are returned by different query engines (via a meta-search-engine). Each document consists of a URL, a document title and a snippet (short text returned by the search engine). Each title and snippet is considered as a bag of words as well.
A Query Memory is a set of queries where each query is modeled as follows:

1- BagTerms: unordered set of terms (bag of words) from the query string;
2- Count: number of times the query was encountered in the query trace;
3- Ldate: last time encountered in the query trace;
4- Fdate: first time encountered in the query trace;
5- QResults: ordered list of URLs and titles returned when the query is submitted, in addition to the snippets (i.e. text that accompany URLs in the result page). The text is modeled as bags of words containing terms from the snippets and title as well as the words in the document path in the URL;
6- Rdate: date the Q*Results* was obtained. Notice that this is the date for the results and it is not necessarily related to *Fdate* and *Ldate*.

The words in *BagTerms* as well as the bag of words associated with the URLs in *QResults* are stemmed using Porter's algorithm [7] and filtered from stop-words. *QResults* elements are composed of: (1) *Rurl*: the URL of the result entry; (2) *Rtitle*: the title of the result entry; and (3) *Rsnippet*: bag of words from either the snippet accompanying the result entry or from the document itself pointed to by the URL.

Using the Query Memory model described above we propose different similarity measures that, given a user query, allow finding all other similar queries from our Query Memory. Notice that an exact similarity is not desired. If a user is unsatisfied with search results and wants hints for queries, if these hinted queries are identical or give an identical result to the original query, the user would not be satisfied. Instead, we want to suggest queries that are close enough in terms of query, or queries that yield results that are comparable in content or description. We have tested different measures for quasi-similarity using either the terms in the query, the terms in the title

or snippet of the search results, or the URL of the search results. Notice that we used the snippets returned by the search engines with the results to represent the document content instead of actually fetch the documents and retrieve the terms from their content. Fetching the documents would necessitate accessing the documents at their respective Web locations and parsing them, which would have added significant overhead. In theory, the real content of a document is the best representative of the document, but we noticed that using the snippets alone was sufficient and lead to acceptable results in a reasonable and practical time. Our model also keeps track of the timestamp when a query is first and last time encountered in the query trace. These timestamps are used in the ranking of similar queries, but they are also used to purge the Query Memory from old queries. It appears that some queries are cyclic (i.e. appearing and disappearing at different periods), or simply periodic (i.e. they appear very frequently but in a short time period, then never again). An illustrative example would be the queries: "terrorism", "El Quaeda" or "Afganistan". They appear after an major event, persist in popularity for some time then fade out when the general interest shifts somewhere else. This is also true with Olympics and other events. Currently, we did not implement the purging of old queries from the Query Memory. However, we use the timestamps for ranking results and introduce the notion of query aging.

3 Overview of Different Similarity Measures

In this section we use the following notation: Q denotes the current query; Δ denotes the Query Memory database; Q.terms indicates the set of terms in the query; Q.Results is the set returned by a search with Q.Results[i] symbolizing the i^{th} entry in the list of results; Q.Results[i].*Title* and Q.Results[i].*Text* represent respectively the set of words extracted from the title and URL, and the set of words extracted from the snippet of the search result entry. We should point out that the notion of query results (Q.Results) is non-deterministic. In other words the same query submitted to the same search engine at a different time can, and usually will produce different results. This is because search engines constantly update their databases and indexes. Some search engines even rotate the order of returned results. For example submitting query "car" twice to the Metacrawler search engine will result in the following three top results respectively for the first and second submission: (1-"Factory Invoice Prices and Dealer Cost", 2-"Free New Car Quotes!", 3-"Car Buying Information") and (1-"Factory Invoice Prices and Dealer Cost", 2-"Shopping for a New Car? Try Invoice-Dealers!", 3-"Classic Cars Photo Ads"). Top results retuned by the same search engine are quite different, even though the same query was submitted to the search engine within a few minutes of each other. In our Query Memory, we do not store all results returned by search engines, but only the first 200 results.

Using the notations described above we define the different quasi-similarity measures as follows:

1- Naïve Query-Based Method:

$$\forall q \in \Delta \,/\, q.\mathbf{terms} \cap Q.\mathbf{terms} \neq \varnothing$$

This is a method used by some existing search engines to recommend expending queries. It simply finds queries with common terms as the current query. There is no word semantics directly or indirectly involved. The results are ordered based on the values of *Count* and *Ldate* (see above). While it is very simplistic, it sometimes yields interesting results. For example "car parts" is found similar to "automotive parts" and "Honda parts". However, it wouldn't be able to find "salsa food" similar to "Mexican recopies". The next method, however, could.

2- Naïve Simplified URL-Based Method:

$$\forall q \in \Delta \,/\, q.\textbf{Results.URL} \cap Q.\textbf{Results.URL} \neq \varnothing$$

This is a method similar to the first one, except instead of looking for common words in the text of a query, we are looking for common URLs returned by the search engines. The reasoning is that if URLs returned by two queries at least have one URL in common, then these queries might be related. All the queries that are found to be similar are then sorted by the occurrence frequencies and are presented in that order. In other words the most common queries that have at least one common URL with the result of the user query are suggested to be similar. This particular method works well on a small dataset of collected queries but has a tendency of making bad suggestions more often than the next method.

3- Naïve URL-Based Method:

$$\forall q \in \Delta \,/\, \theta_m < \frac{\left| q.\mathrm{Results}.URL \cap Q.\mathrm{Results}.URL \right|}{\left| Q.\mathrm{Results}.URL \right|} < \theta_M$$

This method considers the URL set in the search results of the queries. θ_m is a minimum threshold, while θ_M is a maximum threshold. We found that this method could yield interesting results depending upon the thresholds we set. If θ_M is too close to 100%, the queries become too similar. If θ_m is too small, completely different and irrelevant queries could be suggested. In the current prototype implementation we set θ_m to 0.2 and θ_M to 0.8. Method 3 is more specific and more flexible than method 2.

4- Query-Title-Based Method:

$$\forall q \in \Delta \,/\, \exists\, i,\ q.\textbf{Results[i]}.\textit{Title} \cap Q.\textbf{terms} \neq \varnothing \text{ and } q.\textbf{Results[i]} \notin Q.\textbf{Results}$$

This method looks for queries that have in the titles (or URL path) of their results terms appearing in the original query. The idea is that queries that return results with titles related to the original query and the results were not retrieved by the original query could be indeed related to the original query.

5- Query-Content-Based Method:

$$\forall q \in \Delta \,/\, \exists\, i,\ q.\textbf{Results[i]}.\textit{Text} \cap Q.\textbf{terms} \neq \varnothing \text{ and } q.\textbf{Results[i]} \notin Q.\textbf{Results}$$

The idea is similar to the previous method except that the snippets of the results from the candidate queries are considered instead of the titles.

6- Common Query Title Method:

$$\forall q \in \Delta / \exists \text{ i,j, } q.\text{Results[i]}.\textit{Title} \cap Q.\text{Result[j]}.\textit{Title} \neq \varnothing$$

The objective in this method is to compare the terms in the titles returned by the original query with the terms in the titles returned by the candidate queries. No comparison of the query terms per se is done. The main idea of this method is that if two queries return results such that title words in the results returned are similar (there are some common words), then queries are also similar.

7- Common Query Text Method:

$$\forall q \in \Delta / \exists \text{ i,j, } q.\text{Results[i]}.\textit{Text} \cap Q.\text{Result[j]}.\textit{Text} \neq \varnothing$$

This method is similar to the previous one except that the snippets are compared instead of the titles. The last two methods are particularly good at finding queries that are syntactically different but semantically similar. Our experiments showed that on a small set of processed queries this method performs better than the previous one, however it is in an order of magnitude more computationally intensive than the previous one. Selecting the content of pages instead of the snippets returned by the search engines would make the computation is more time-consuming.

For all these methods, results are sorted. The ranking of the suggested queries is based on either the *Count* and *Fdate*, or the number of common URLs. Only a limited number of quasi-similar queries are presented to the user. Notice that we can easily combine these measures, in particular 3 and 4, or 5 and 6. The last two similarity measures are particularly computationally expensive and their time complexity is proportional to the size of the collected database. We are considering new indexes to reduce the time it takes to recommend queries with measure 6 or 7.

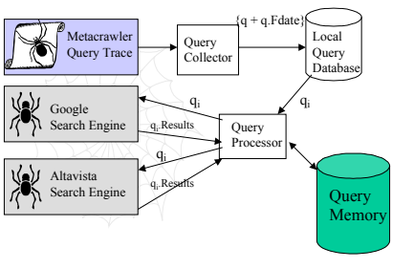


Fig. 1. Constructing the Query Memory

4 Prototype Implementation Overview

Figure 1 illustrates the general architecture of our prototype constructing the Query Memory. A query collector collects queries from a popular metacrawler and stores them in a query database with a timestamp. Query processors submit these queries to

other search engines to collect results and store these results in the Query Memory with updated counts and timestamps. Different search engines are used to harvest search results and the original metacrawler is not used to submit queries in order to avoid affecting the query trace of the metacrawler, and thus avoid compromising the counts of the queries in our Query Memory by processing collected queries.

We have implemented our prototype on NetBSD, an operating system allowing large disk partitions, and MySQL database. The query database collected quickly reached more that 2 Gigabytes, the largest disk partition allowed by Linux. The collection and processing of the queries is implemented with Python scripting language, while the similarity engine is in C++. Figure 2 shows an overview of the query recommendation phase.

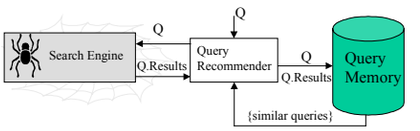


Fig. 2. Query quasi-similarity engine

Our system consists of three major parts: a web query collector, a web query processor (Figure 1) and a quasi-similarity engine (Figure 2) that finds similar queries among previously collected queries to the one specified by the user. The query collector simply collects queries submitted to the Metacrawler search engine[1] and puts them in the query queue along with the timestamp when these queries were obtained. The Metacrawler search engine provides its query trace by displaying at regular times a list of currently submitted search queries[2]. The query collector agent is simply a Python script that at regular time intervals consults the Metacrawler query trace provided and parses it to extract the query strings. The query processor verifies for each query if it already exists in the Query Memory, as described above. If the query already exists in the Query Memory, then it just updates the relevant counters and timestamps. Otherwise, the query is submitted to the Google search engine and AltaVista search engine to harvest the corresponding search results. These results are then parsed, processed, and added in the Query Memory.

The quasi-similarity engine is triggered whenever a suggestion for queries is requested. The user specifies which similarity measure he or she wants to use to find similar queries. The similarity engine uses previously collected queries stored in the Query Memory to return suggested similar queries. We implemented a basic web-based interface that allows the submission of a web search query. The submitted query is sent to Metacrawler and the search results are displayed. If the user is unsatisfied with the results a recommendation for similar queries is requested and similar queries are displayed based on the similarity measure chosen. Because of lack of space, we are not showing a snapshot of our implemented prototypical user interface. The entire web page of the prototype is divided into 4 frames. The top frame allows the user to specify a query, which is sent to the Metacrawler search engine. The results returned

[1] MetaCrawler: http://www.metacrawler.com

[2] MetaSpy: http://www.metaspy.com/spymagic/Spy?filter=false

are displayed integrally in the results frame (second from the bottom). Note that Metacrawler was arbitrarily selected for our prototype. Other search engines could be used or the choice could be given to the user. If the user is not satisfied with the results, he or she can request similar queries after selecting a quasi-similarity measure in the second frame. Suggested new queries are displayed in the bottom frame along with hyperlinks that would automatically submit the query if clicked, thus allowing an interactive process of query refinement.

5 Discussion on Some Experimental Results

This section presents some examples of similar queries using our measures of quasi-similarity for illustration purposes. We have implemented in our prototype the quasi-similarity measures 1, 2, 3, 6 and 7 as presented in Section 3. Using the interface described in Section 4, we submitted many queries and checked the suggested similar queries based on all implemented measures. While the similarity measures are different, it was common that the recommended queries with the different measures were alike. Moreover, it was difficult to find a winner (i.e. the similarity measure with the best recommendation) because: (1) for each different query we experimented with, a different winner could be proclaimed; (2) the search results and the recommendation are nondeterminstic (i.e. we can expect different results for the same query submitted twice); and (3) the validation is very subjective. The results of the query recommender, however, are very encouraging. Below we present two of our results randomly selected from the different experiments we performed: "salsa food" and "computer prices". Tables 1 and 2 do not contain all recommended queries but a top ranked selection. We found indeed queries such as "hot girls", latina porn", or "BBC News" as matches to the "salsa food" query for example. The match, however, is understandable. First these queries were submitted by someone, and thus were in the query trace. Second, there is a connection between "salsa", "hot" and "latina" indirectly discovered using our quasi-similarity measures by intersecting the search engine results. The "BBC News" match, on the other hand, is more difficult to explain. It is possible that at the time the "BBC News" query was processed, the results could have contained links to recipes, hence the intersection with salsa food. This also demonstrates the difficulty to use precision and recall as validating measures, given the subjectivity of the similarity.

Table 1. Recommended quasi-similar queries for the query "salsa food"

Query:	"salsa food"
Measure 1:	Food Recipes, Japanese Food, genetically engineered food, food pictures
Measure 2:	recipes, Mexican Recipes, All Recipe.com, french cooking
Measure 3:	recipes, Mexican Recipes, All Recipe.com, french cooking
Measure 6:	junkfood, foods history, black pepper health benefit, Food Cooperative
Measure 7:	peruvian dry rubs, green curry recipes, black pepper health benefit, australian foods

Table 2. Recommended quasi-similar queries for the query "computer prices"

Query:	"computer prices"
Measure 1:	car prices, computer hardware, computer dictionary, airline ticket prices, computer parts
Measure 2:	computer memory, consumer price index, toms hardware, cheap computer upgrades, Hardware Stores
Measure 3:	price search, price compare, pricewatch.com, toms hardware, low cost computers, Wholesale Computer Pricewatch
Measure 6:	buying laptops, IBM Aptiva, online trading, ps one memory card, Shopping
Measure 7:	buying laptops, bid software, DVD Player & Review, christmas online shopping

While measure 1 is very simplistic and matches only queries with terms in the original query, some of the results are nevertheless relevant to some extent. Measures 2 and 3, in the case of "salsa food", returned almost the same results; which is to be expected, since we do not have many queries processed in our Query Memory database yet. Therefore we set the threshold parameter θ_m to fairly a low level of 0.2. Measures 6 and 7 are looking for the common words in query titles or snippets and are capable of suggesting more "diverse" queries. The later measure, however, seems to make more relevant suggestions than the former one. This should not be a big surprise, since it looks for common words in snippets that are more descriptive than the title pages. On the down side, it is computationally very expensive.

There are two major observations from our experiments with our proposed quasi-similarity measures. First, the more queries we process, the richer the Query Memory is, and the better the recommendations are. While we have collected more than half a million different queries from the Metacrawler search engine, we have processed and harvested the results of only about 70,000 queries (which constitutes the Query Memory). Submitting queries to the search engine and harvesting result is very computationally and network intensive process. Ideally, if the query recommender is on the search engine side, not only the query trace would be immediately available, but also the inverted indexes of the search engine would also be available avoiding the submission of queries for results harvesting. Also because of the limitations of the MySQL database and the file systems file size problems, it would be better to directly process queries collected and populate the Query Memory rather than storing collected queries in a large queue. Only queries in the Query Memory can be used in the similarity measures. The second observation is that it seems difficult to select the best method. Apparently some combination of different methods with weights assigned might produce better results than every single similarity taken alone. This is another open problem that we will be addressing in our future research.

6 Conclusion and Remarks

We have implemented a recommender system to suggest similar queries for search engine users when they are not satisfied with the result of an original query and would

like help in expanding or improving the search query terms. We proposed some quasi-similarity measures to ensure that the suggested queries are not too similar to the unsatisfactory original query but analogous enough to be suitable. We have experimented with all these measures isolated or combined, and have noticed that it is very difficult to find an absolute winner. Each quasi-similarity measure could indeed generate very good results depending upon the query. In some cases what we expected to be the winner actually produced irrelevant queries. We have thus kept all the measures and left it to the user to select and experiment with the desired methods. It is important to notice that since the Query Memory is updated in real time by adding new queries, and since some queries could be removed if considered too old based on *Ldate*, the recommendation is never deterministic. In other words, the system, for the same original query, could suggest different similar queries at different times. In the current work we are also investigating the use of time constraints to limit the queries to those submitted in a time range. This is useful because queries could be time sensitive, appearing frequently at one point than disappearing completely later. We are also investigating the use of Latent Semantic Indexing [3] for finding relevant associations between titles or text returned in the search engine results. Finally, we are analysing the possibility to validate these quasi-similarity measures using precision and recall, and studying the effect of purging old queries from the Query Memory on the precision and recall.

References

1. Junghoo Cho, Hector Garcia-Molina and Lawrence Page, Efficient Crawling Through URL Ordering, 7th International Conference on WWW, Brisbane, Australia, 1998
2. Sergey Brin and Lawrence Page, The Anatomy of a Large-Scale Hypertextual Web Search Engine, 7th International Conference on WWW, Brisbane, Australia, 1998
3. S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer and R. Harshman, Indexing by Latent Semantic Analysis, in Journal of the American Society of Information Science, 1990
4. Fitzpatrick, Larry and Dent, Mei 1997 Automatic Feedback Using Past Queries: Social Searching? In Proeedings of SIGIR'97, Philadelphia, PA.
5. Natalie S. Glance, Community search assistant, Workshop on Artificial Intelligence for Web Search, In conjunction with the AAAI conference, Austin, Texas, USA, 2000
6. Evangelos P. Markatos. On Caching Search Engine Results. In Technical Report 241, ICSFORTH, January 1999. available at http://archvlsi.ics.forth.gr/html_pages/TR241/
7. M. F. Porter, An algorithm for suffix stripping, Program, pp 130-137, vol 14, n 3, 1980.
8. Danny Sullivan, Search Assistance Features: Related Searches, 2001 available at: http://www.searchenginewatch.com/facts/assistance.html

WOnDA: An Extensible Multi-platform Hypermedia Design Model

Dionysios G. Synodinos[1] and Paris Avgeriou[2]

[1]National Technical University of Athens, Network Management Center
Heroon Polytechniou 9, Zografou 157 80, Greece
dsin@noc.ntua.gr
[2]National Technical University of Athens, Software Engineering Laboratory
Heroon Polytechniou 9, Zografou 157 80, Greece
pavger@softlab.ntua.gr

Abstract. The design and development of hypermedia applications that are deployed on the Web and other delivery platforms is largely conducted on an intuitive, ad hoc basis, thus resulting in inefficient systems that are hard to modify, maintain and port to alternative platforms. There are now justifiable research and development efforts that attempt to formalize the engineering process of such systems in order to achieve certain quality attributes like modifiability, maintainability and portability. This paper presents such an attempt for designing a conceptual model of a hypermedia application that allows for easy update and alteration of its content as well as its presentation and also allows for deployment in various platforms. In specific this model explicitly separates the hypermedia content from its presentation to the user, by employing XML content storage and XSL transformations. Our work is based upon the empirical results of designing, developing and deploying hypermedia applications in various platforms, and on the practices of well-established hypermedia engineering techniques.

1 Introduction

During the last decade web pages have evolved to a point of becoming too complex with all the inline client scripts (e.g. JavaScript) and styles rules in order to facilitate the ever-growing and often conflicting demands for enhanced usability and impressive "look and feel". Therefore the daily task of updating content can no longer be performed by a novice in HTML, but instead designated professionals with a solid background on web authoring and a clear understanding of the architecture of the certain site must be utilized.

To make matters worse, even though the functional and non-functional requirements of web sites can be satisfied through the use of standard technologies like the latest versions of HTML and JavaScript, web authors find themselves using

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 217-228, 2002.

more and more proprietary solutions to facilitate customer needs and overcome the weak compatibility in this area, that major browsers offer. This situation can lead to the need of internal branching in sites that heavily deploy cutting-edge DHTML or browser specific code. Thus the number of maintained templates grows and the administrative tasks mentioned above can become overwhelming.

On top of everything else, alternative versions of web sites have become common practice nowadays to satisfy technological, political, economic and social goals. These alternative versions include mobile devices sites, accessibility sites, printable versions of sites, voice-enabled sites.

The World Wide Web Consortium reacted quite early to the above, accumulated problems with the launch of XML and the related family of technologies, in order to separate the content from the rest of the information such as presentation rules, metadata, active components etc. The question now is, given the XML technology, how can a site be engineered in order for it to achieve modifiability, maintainability and portability.

In this paper we attempt to solve the above problems by proposing an XML-based multi-tier model, named **WOnDA** (**W**rite **On**ce, **D**eliver **A**nywhere), which is established upon the separation of the actual content, active widgets, presentation rules and the page generation process. The presentation rules are themselves separated into a set of rules for transforming the actual content, the various widgets and the layout of the pages for every one of the presentational domains. Furthermore for domains that draw upon HTML (web browser versions) there is an option of defining one more presentational layer that lays above all the rest and utilizes the W3C's Cascading Style Sheet (CSS) specifications.

The proposed model is based on an XML repository that holds the actual (textual) content and utilizes the power of eXtensible Style Sheet (XSL) transformations in a hierarchical manner for providing a rich set of formats to deploy content It also facilitates effortless administration, the ability to easily add new formats and fast/clear refactoring of old ones. Also, since there is a complete separation between data and presentation, the development of content can become a streamlined process that doesn't deal with the complexity of the underlying structure of the chosen presentational domains.

The structure of the rest of this paper is as follows: Section 2 introduces a short literature review, comprised of the most significant approaches in designing hypermedia applications. Section 3 analyses the proposed model and its philosophy. Section 4 presents a complete case study of a single page that is created with the aid of the model and deployed in three platforms. Finally section 5 wraps up with ideas for future work.

2 Literature Review

During the past years there have been several attempts to formalize processes, models, methods, techniques and best practices for developing hypermedia applications [1].

The **Dexter Hypertext Reference Model** [2] was perhaps the first successful attempt at modeling hypermedia applications and has been the predecessor and source

of inspiration for many of the approaches that followed. Another widely used approach of hypermedia design modeling is **HDM** [3] and its successor HDM2 [4], where emphasis is given in specifying a semantic schema of the application before it is developed. Its main feature is the definition of a number of dimensions, along which a hypermedia application can be modeled: structure, navigation, behavior, user control and presentation. The **Object-Oriented Hypermedia Design Model (OOHDM)** [5] is based on the separation of the hypermedia application's process into four phases: Conceptual Design, Navigation Design, Abstract Interface Design and Implementation. A CASE Tool named OOHDM-Web [6], allows implementation of hypermedia applications as CGI scripts that produce dynamically generated pages, whose contents are fed from a database and integrated with predefined templates. Another methodology that provides step-by-step hypermedia development is **Relationship Management methodology** (RMM) [7]. RMM offers complete representation of the semantic schema and the navigational schema, follows the traditional E-R model to standardize the conceptual and navigational design but gives limited support to the interface design.

The **CC/PP framework** [8] of the World Wide Web Consortium aims to provide a common way for clients to express their capabilities and preferences to the server that originates content. WOnDA can be intergraded with the CC/PP model in the manner that different presentation formats can be created in an asynchronous way to satisfy the device profiles collected from a central or various distributed profile registries. But since WOnDA proposes that, creation of hypermedia context is done prior to user request (static content), a scenario where the content is adapted dynamically to the capabilities of the device by reading the device's profile in real time, is not possible. The later though can be helpful in a database-driven website. Also an important approach is **Sisl** [9], which is an architecture and domain-specific language for designing and implementing services with multiple user interfaces. It aims to the decoupling of interface from service logic, by employing an event-based model of services that allows service providers to support interchangeable user interfaces to a single source of service logic or data.

3 The WOnDA Model

A macroscopic view of the model's functionality is illustrated in Figure 1. The content is authored in an editor that could be a simple text editor or any contemporary word processor like MS Word or Word Perfect. It may include text and references to other media such as images, video, sound, animation etc. The content then is translated into an XML file that properly describes its various parts like text, hyperlinks, video clips etc. In sequence, XSL transformations are utilized to impose style rules and presentation layout, add active objects, and generate the page in its final form, e.g. HTML page, WML page etc. The final page is then served to the appropriate client through the Internet by being published to a Web Server.

The detail that is missing from the above figure, is the mechanism that deals with the XML files and XSL transformations that translate raw content into a specific delivery platform. This mechanism is described in the rest of this section and aims to

serve as a guide for the construction of maintainable, modifiable, and portable hypermedia applications.

In order to describe the model we utilize the Unified Modeling Language [10] (http://www.rational.com/uml), a widely adopted modeling language in the software industry and an Object Management Group standard [http://www.omg.org/]. Furthermore in order to define the model we have designed a UML meta-model that helps define what is a well-formed model – i.e. one that is syntactically correct.

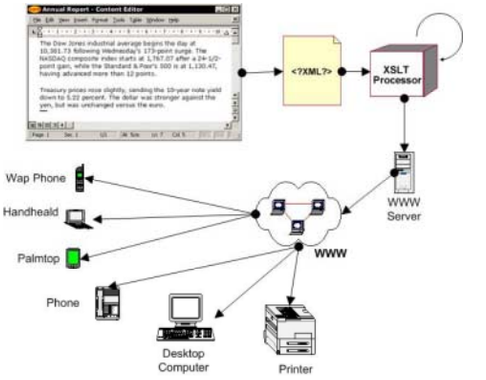


Fig. 1. A macroscopic view of the model

The model described here considers only static hypermedia pages and every page is comprised of the following elements:

1. The actual content of the page
2. A set of navigational or promotional active objects or widgets like navigation bars, search boxes, menus, logos, ads, banners etc.
3. The general layout of the page meaning the positioning of all the above in the browser window and the rest of the markup envelope that is needed in order for the page to be syntactically valid.
4. Hyperlinks to other hypermedia pages

It is noted that this is a simplified and superficial model of a hypermedia page because the aim of the model is not to model hypermedia applications in general but merely to separate content from the rest of the information and generate multiple versions of hypermedia applications. In other words the proposed model is considered to be in a lower abstraction layer than usual hypermedia design models such as HDM [3], OODHM [5], RMM [7], WebML [11] etc.

The principles of the proposed meta-model are the following:

1. The actual content of each hypermedia page is kept in one XML file. These files will be referred to as **Content Pages** (CPs). CPs represent published pages as abstract data entities without taking into account any presentation aspects derived by the desired formats.
2. The task of providing content rendering information is left up to a set of XSL files, which will be referred to as **Content Transformers** (CTs). If we define a set of N versions for the site under construction, every version is exactly identical to all the others in terms of textual information since this information is provided by the CPs, but the versions differ in the layout, functionality, style and the markup that they're written in. For every one of these versions we define a CT which describes the rules necessary to transform the content provided by the respectable CPs.

3. In the fashion of CPs and CTs for the textual content we define **Widget Pages** (WPs) and **Widget Transformers** (WTs), which hold the necessary data and presentation rules respectively for the widgets used.
4. Metadata that are specific to the content page, as in the HTML <META> element, used throughout a version or even throughout the entire site are kept in an XML file, which will be referred to as **Content Page Metadata** (CPM).
5. Metadata that are specific to a version, e.g. character encoding information, are kept in another XML file, called **Version-Specific Metadata Page (VSMP).**
6. Both content page-specific and version-specific metadata are rendered by another transformer called **Metadata Transformer (MT).**
7. For every one of the different versions we define an XSL file, which describes the rules necessary to generate the page layout that is restricted in the context of the version. These XSL files, which will be referred to as **Version Builders (VBs)**, do not contain any information about the rules we need to render the textual content drawn from the CPs nor the widgets used. They rather define the general layout of the page meaning the positioning of all the above in the browser window and the rest of the markup envelope that is needed in order for the page to be syntactically valid for the corresponding presentational domain. The information about client scripts and CSS, are referenced by the VB, where needed and are imported in the document e.g. via the HTML <LINK> element. Figure 2 depicts the relationship between the above model elements. Content Pages, Widget Pages and Content Page Metadata are XML files and are all specializations of the class "Generic Hypermedia Page Element". They are also connected with an aggregation relationship with the "Hypermedia Page" class, which means that they are all part of a hypermedia page. Content Transformer and Widget Transformer are XSL

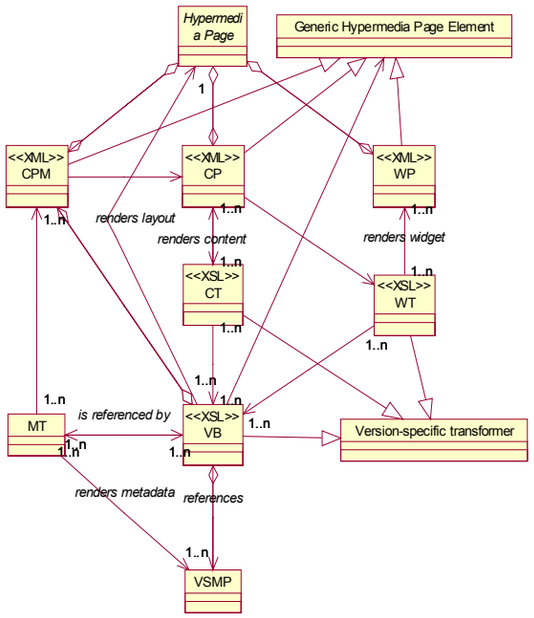


Fig. 2. The hypermedia page elements and the transformers

files that render the corresponding Content Pages and Widget Pages. Furthermore it is obvious that Content Metadata Pages are related to Content Pages, in the sense that metadata describe the content. Moreover, Content Metadata Pages and Version-Specific Metadata Page are rendered by the Metadata Transformer. Finally Version Builders use all the other transformers to render the layout of the hypermedia page and insert the appropriately transformed content, widgets and metadata into them. Content Transformers, Widget Transformers and the Version Builder are specializations of the "Version-Specific Transformer" class.

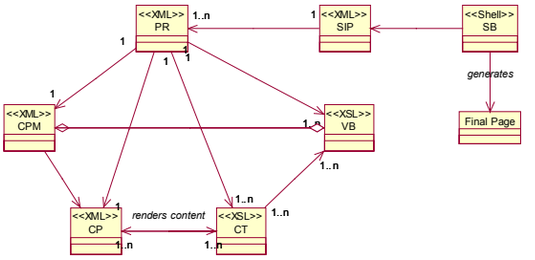


Fig. 3. The Page Generation Process

8. For every site there is a main registi main processing mechanism. This master r which holds all the file system (or networl ne of them to a registry specific for it, called **Page Registry (PR)**. PRs link together CPs and PTs for the various versions.
9. All the above are parsed by a processing shell, which will be referred to as ***Site Builder*** and provides the web site administrator with a web interface to generate or update certain pages, entire versions of the site etc.

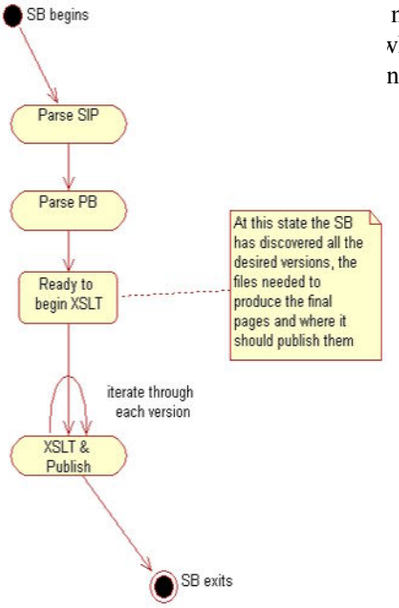


Fig. 4. The functionality of the SB through a UML activity diagram

Figure 3 depicts the page creation process that takes place with the aid of the last three elements. The Page Registry gathers all the necessary data from the Content Page, the Content Transformer, the Content Page Metadata and the Version Builder. The Site Builder parses the Site Index Pages to look for all the Page Registries, performs the transformations and generates the hypermedia page of the appropriate format, as illustrated in Figure 4 through a UML activity diagram.

4 A Case Study

In order to explicate WOnDA, we'll examine in detail the design of a web page of a laboratory that offers references to external web sites holding tutorials on XML. This page should be accessible from the Mozilla browser, from a normal WAP phone and we must also provide it in an easily readable format for students with disabilities. Figure 5 depicts the final product in the desired formats. Figure 6 depicts the implementation of the meta-model described earlier in the case of the "XML Resources" page. This is an instance of the meta-model, i.e. a model per se that is consisted of instance of all the necessary meta-model elements. It is noted that since this example is rather simple, not all the meta-model elements are needed to describe it.

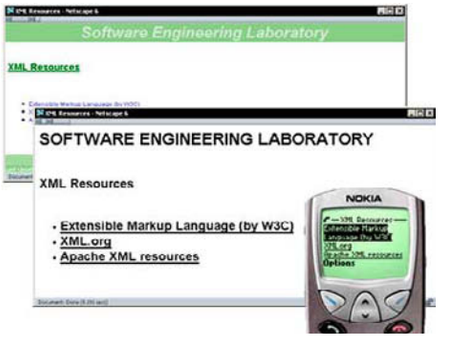


Fig. 5. The three different version of a laboratory web page

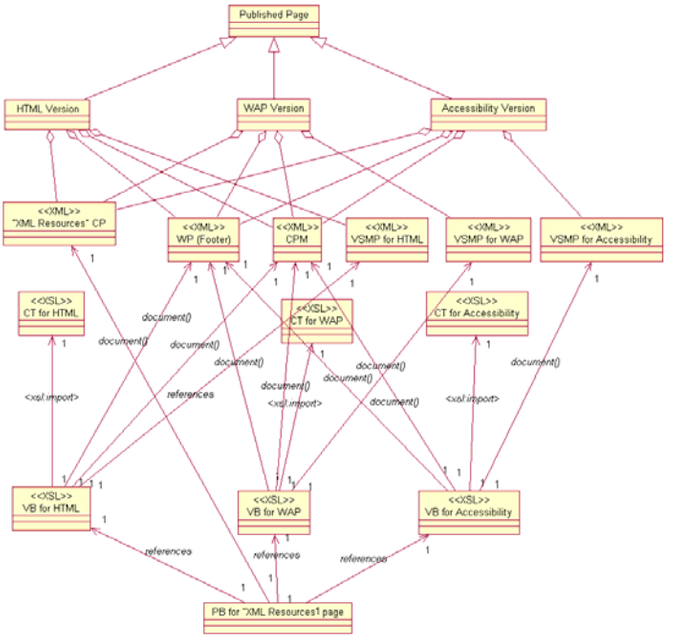


Fig. 6.

1 Content Pages (CPs)

As mentioned earlier the actual content of a page is described inside a CP. The following code listing holds both the textual information of our "XML Resources" page and the needed references to external resources like the external links. The XML vocabulary used here is pretty self-explanatory since it uses widely recognized terms like "paragraph", "list" etc. for the names of the elements used.

Example CP for the "XML Resources" page

```
<data>
<title value="XML Resources"/>
<content>
<paragraph>
  <emphasize>XML Resources</emphasize>
</paragraph>
<list type="unordered">
  <ListElement>
    <link location="http://www.w3.org/xml">
      Extensible Markup Language (by W3C)
    </link>
  </ListElement>
  <ListElement>
    <link location="http://www.xml.org">
      XML.org
    </link>
  </ListElement>
  <ListElement>
    <link location="http://xml.apache.org">
      Apache XML resources
    </link>
  </ListElement>
</content>
</data>
```

2 Content Transformers (CT)

The CTs for our page will provide the processing shell (SB) the necessary rules in order to transform the XML code of the CP. The final form of such a CT depends on the elements used inside the CP. In the following code listing we suggest a set of four XSL rules to handle the transformation of the elements <paragraph>, <list> and <link>. These transformations will produce an HTML fragment that is suitable for the Mozilla browser version of the page.

Example CT for the "XML Resources" page

```
<xsl:template match="paragraph">
  <p><xsl:apply-templates/></p>
</xsl:template>

<xsl:template
match="list[@type='unordered']">
  <ul><xsl:apply-templates/></ul>
</xsl:template>
<xsl:template
match="list[@type='unordered']/ListElement"
>
  <li class="UnorderedListElement">
<xsl:apply-templates/></li>
</xsl:template>
```

```
<xsl:template match="link">
  <a href="{@location}"><xsl:apply-
templates/></a>
</xsl:template>
```

3 Widget Pages (WP) and Widget Transformers (WT)

The paradigm of CPs and CTs is also used for the WPs and the WTs as the code listing on the right shows. This is an example of XSL transformations for the WP for the "XML Resources" footer as described above, for the Mozilla version.

```
<widget name="Footer">
  <element type="TextLink"
    label="Homepage" link="CP.Home.xml"/>
  <element type="TextLink"
    label="Contact" link="CP.Contact.xml"/>
  <element type="TextLink"
    label="Search" link="CP.Search.xml"/>
  <element type="TextLink"
    label="Webmaster" link="mailto:webmaster@lab"/>
</widget>

<xsl:output encoding="ISO-8859-1"/>
<xsl:template match="widget[@name='Footer']">
  <p><xsl:apply-templates/></p>
</xsl:template>

<xsl:template match="element">
  &amp;nbsp;<a href="{@link}">
    <xsl:value-of select=".[@label]"/>
  </a> &amp;nbsp;
</xsl:template>
    </xsl:stylesheet>
```

4 Metadata Pages (MP)

For the purpose of supplying our page with the required meta-data we use both the CPM and the VSMP mechanism described earlier for the model. In the following code sample we describe a way to handle three trivial cases of meta data information that appears in web pages.

Example nodes for Metadata pages either CPM or VSMP

```
<MetaDataElement type="author"
  content="Paris Avgeriou"/>

<MetaDataElement type="http-equiv"
  value="Expires"
  content="Tue, 20 Aug 2002 14:25:27 GMT"/>
```

5 Version Builders (VBs)

The VB of every version will import the XSLT rules described in the CT, WT and MT of the later version and

The skeleton of the VB for the Mozilla browser

```
<xsl:import href="CT.Web_Mozilla.xsl"/>
<xsl:import href="WT.Footer.Web_Mozilla.xsl"/>
```

provide the overall envelope to bind rules and content. The following code sample is a simple VB for the Mozzilla browser.

Key elements of the XSL sample on the right are the imports of the other XSL rules, in the <xsl:import> tag, and the use of the document() function to include any other XML document besides the CP.

```
<xsl:import href="WT.Footer.Web_Mozilla.xsl"/>
<xsl:variable name="Footer"
  select="document('WP.Footer.xml')"/>
<xsl:variable name="CPM"
  select="document('CPM.XMLResources.xml')"/>
<xsl:variable name="VSMP"
  select="document('VSMP.Web_Mozilla.xml')"/>
<xsl:output encoding="ISO-8859-1" />
  <xsl:template match="/">
  
  </xsl:template>
    </xsl:stylesheet>
```

6 Site Index Page (SIP) and the Page Registry (PR)

In the following part we give an example of a SIP and a PR. The syntax is again self-explanatory.

```
<pages>
  <page description="foo"
    PR_Path="PR.foo.xml"/>
  <page description="bar"
    PR_Path="PBR.bar.xml"/>
  <page description="XML Resources"
    PR_Path="PBR.XmlResources.xml"/>
</pages>

  <ContentPage path="CP.XmlResources.xml"/>
  <version name="Web_Mozilla"
    CT_Path="CT.Web_Mozilla.xsl"
    Pub_Path="/www_root/html/XMLResources.html"/>
  <version name="WAP"
    CT_Path="CT.WAP.xsl"
    Pub_Path="/www_root/wml/XMLResources.wml" />
  <version name="Accesibility"
    CT_Path="CT.Accesibility.xsl"
      Pub_Path="/www_root/accs/XMLResources.html" />
```

7 Processing Shell (Site Builder - SB)

As mentioned above the entity responsible for applying the final XSL Transformation and publishing the pages is the Site Builder. This module was implemented as a Java 2 Application that consists of a web-based front-end for the administrator and an internal mechanism to apply XSLT using any Java-XML package.

5 Future Work

Although the principles described above provide a solid foundation, they do not deal with the entirety of the diverse scenarios that are popular in contemporary hypermedia applications. To better handle this without affecting the simplicity of WOnDA, which

is one of its strongest attributes, we plan to extend this model in a modular way having different *flavors* of the original approach, such as:

- **Distributed flavor**: In cases of web sites of large organizations or multinational corporations, the operations of content authoring, style development, page generation and publishing can be executed in a distributed way, over a plethora of locations worldwide, in a pattern designated by the business plan or the established workflow of the respective organization or corporation.
- **Multilingual flavor**: multilingual context should be better handled.
- **Database & server side scripting flavor:** the model should deal specifically with situations where the developer needs to systematically draw information from a database or merely execute HTML-embedded server side scripts.
- **User personalization flavor:** the model should take into account the practice of dynamically adapting pages according to the site visitors.

Finally, an interesting issue that is under research is the way that WOnDA can benefit from all the work recently done in the database area in order to deal with XML documents. This is caused by the variety of proposed models for storing and accessing XML information both natively and on object-relational databases. To examine this issue more closely the authors are actively involved with "INEX: Initiative for the Evaluation of XML retrieval" [http://qmir.dcs.qmul.ac.uk/INEX/].

References

1. Lowe, W. Hall, Hypermedia & the Web, an Engineering Approach, John Wiley & Sons: 1999.
2. F. Halasz and M. Schwartz. "The Dexter Hypertext Reference Model". *Communications of the ACM*, 37(2):30-39,Feb.1994
3. F. Garzotto, D. Schwabe, P. Paolini, "HDM- A Model Based Approach to Hypermedia Application Design", ACM Transactions on Information Systems, Vol. 11, #1, Jan. 1993, pp. 1-26.
4. F. Garzotto, L. Mainetti, P. Paolini, "Navigation in Hypermedia Applications: Modeling and Semantics", Journal of Organizational Computing and Electronic Commerce. 6, 3 (1996), 211-238.T.
5. D. Scwhabe & G. Rossi, "The Object-Oriented Hypermedia Design Model (OOHDM)", Communications of the ACM, Vol. 35, no. 8, August 1995.
6. D. Schwabe & R. de Almeida Pontes, "OOHDM-WEB: Rapid Prototyping of Hypermedia Applications in the WWW", Tech. Report MCC 08/98, Dept. of Informatics, PUC-Rio, 1998.
7. T. Isakowitz; E. Stohr; P. Balasubramaniam, "RMM, A methodology for structured hypermedia design", *Communications of the ACM*, August 1995, pp 34-48
8. World Wide Web Consortium (W3C), "Composite Capabilities/Preference Profiles: Requirements and Architecture", W3C Working Draft 28 February, 2000.

9. T. Ballz, C. Colby, P. Danielseny, L. Jategaonkar Jagadeesany, R. Jagadeesan, K. Läufer, P. Matagay, K. Rehory, "Sisl: Several Interfaces, Single Logic", Journal of Speech Technology, Kluwer Academic Publishers, 2000.
10. G. Booch, J. Rumbaugh, and I. Jacobson, The UML User Guide, Addison-Wesley, 1999.
11. S. Ceri. P. Fraternali, A. Bongio, "Web Modeling Language: a modelling language for designing Web sites", WWW9 Conference, Amsterdam, 5/2000.

Model-Driven Approaches to Software Development

Dan Turk[1], Robert France[1], Bernhard Rumpe[2], and Geri Georg[3]

[1] Colorado State University, USA
[2] Technische Universität München, Germany
[3] Agilent Technologies, USA

Preface

"Extreme Programming", "Agile Modeling", "Model-driven software development", "Model-driven architectures", "Agile Development", "OMG's MDA". These catch-phrases are currently the topic of much discussion in the software development world.

Some software methodologists say that modeling should drive the whole development process. Some say modeling should be only done informally, and that models should not be kept for future use (i.e., that they are temporary artifacts produced during development). How useful is modeling? When should it be done? Can modeling be demonstrated to provide business value? Can architectures be developed using "agile" or XP approaches? What makes an approach agile, and what makes agile approaches advantageous? Can model-driven approaches be agile? Can we quantify the benefits of agile approaches?

These are all questions currently receiving attention in the software development world. Participants in this workshop discussed these, and related topics such as techniques integrating agile approaches and model-driven development. Papers focusing on the integration (or lack of ability to integrate) of agile and model-driven approaches fit well within the OOIS conference theme, "Information System Integration."

Relevant topics of the workshop included:

1. Issues in model-driven development
2. OMG's Model-Driven Architecture (MDA)
3. The value of models (e.g., business models, enterprise architectures, requirements and design models)
4. Modeling techniques and notations relevant to IS development
5. Assumptions underlying model-driven development
6. The compatibility (or tension) between agile and model-driven development
7. Business value of modeling
8. Empirical validation of the value of modeling, especially using the OMG's UML and MDA
9. Empirical support for the effectiveness of model-driven and agile development approaches
10. Empirical validation of the benefits of Agile development (XP, Scrum, etc.)

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 229–230, 2002.

Organization

This workshop is organized by Robert France, Colorado State University, Fort Collins, Colorado, USA, Geri Georg, Agilent Technologies, Fort Collins, Colorado, USA, Bernhard Rumpe, Munich University of Technology, Munich, Germany and Dan Turk, Colorado State University, Fort Collins, Colorado, USA.

Program Commitee

J. Bezivin, University of Nantes, France.
T. Clark, King's College, London, UK.
A. Evans, University of York, UK.
R. France, Colorado State University, USA.
G. Georg, Agilent Laboratories, USA.
A. Moreira, New University of Lisbon, Portugal.
B. Rumpe, Munich University of Technology, Germany.
R. Trask, Qwest, Ohio, USA.
D. Turk, Colorado State University, USA.

Primary Contact

For more details on the workshop please contact:
Geri Georg
Agilent Laboratories
Agilent Technologies
Fort Collins, Colorado, USA

See workshop website below for additional information:

`http://oois-mdsd.cs.colostate.edu`

Executable and Symbolic Conformance Tests for Implementation Models (Position Paper)

Thomas Baar

Universität Karlsruhe, Fakultät für Informatik
Institut für Logik, Komplexität und Deduktionssysteme
Am Fasanengarten 5, D-76128 Karlsruhe
baar@ira.uka.de

Abstract. Following the Model-Driven Architecture (MDA), a system description consists of several models, i.e. views of the system. This paper is concerned with formal conformance tests between different models. It stresses the need for formal semantical foundations of all languages that are used to express models. In particular, we classify conformance tests for implementation models.

The Model-Driven Architecture (MDA) [14,2] is the new strategy of the OMG to maintain the whole lifecycle of a system. A system is specified by a collection of several views, which concentrate on certain aspects and hides all non-relevant details. In the terminology of MDA a view is called *model*. Different kinds of models are usually expressed in different suitable languages. The implementation of a system is seen as just one model among others. MDA does not stipulate the usage of particular languages to express particular models. However, implementation models are usually expressed in terms of a programming language.

Especially in the development phase of a system, periodical checks on the consistency of all models are crucial. MDA calls this activity *conformance test* between two or more models. In this paper, we focus on conformance tests which can be performed by a machine. Obviously, such conformance tests presuppose a formal underpinning of all languages used to express the involved models.

We concentrate on the conformance test between the implementation model and a single additional model. For practical reasons, we assume Java to be the language for expressing the implementation model. The semantics of Java was originally defined in [3]. Due to its informal nature, [3] cannot be used directly for machine-based conformance tests. Instead, another representation of Java's semantics has to be chosen. We discuss two representations, and investigate the influence of that choice on the nature of the corresponding conformance tests.

Java Semantics Given by a Compiler A pragmatic and popular approach defines the semantics of Java by the actual behaviour of programs on a machine. Adopting this approach for the semantical foundation of implementation models means to define implementation models in terms of the

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 231–234, 2002.

behaviour of a *Black Box* that consists of the Java byte-code compiler and the Java Virtual Machine (JVM). This is fully acceptable but imposes a serious restriction for conformance tests. Obviously, conformance tests can be only performed against those models which rely semantically on the same *Black Box* and therefore are written in Java as well.
For large-scale projects, the development of unit tests [6] has proven to be a good practice. A suite of unit tests can be considered as a model which is different from the implementation model but written in the same implementation language. For instance, let us consider the following implementation:

```
public class Foo{
    public int addTwo(int i){
        return i+2;
    }
}
```

Assume, we want to test the conformance between the implementation model of class `Foo` and another model, that requires the result of operation `addTwo` to be always greater than the argument. The intended model can be (partially) expressed by some unit tests, e.g.

```
aFoo.addTwo(3) > 3
aFoo.addTwo(5) > 5
```

The conformance test between both models is done simply by the automatic execution of the two test cases. Therefore, we name such conformance tests *executable conformance tests*.

The example reveals the big disadvantage of models given in a unit-test style. Here, the language Java is abused as a specification language with very limited expressive power, i.e. Java is unable to express abstract properties of a system. In the above example, the intended requirement – the return value of `addTwo` is always greater than the argument – can only be formulated approximately, by asserting it for a few arguments.

Abstract models, i.e. models which describe more abstract properties of the system, are therefore better formulated using a more suitable specification language than Java, e.g. OCL [13] or JML [8]. However, at a first glance, we have to sacrify the conformance tests between abstract models and implementation models because the semantical descriptions of the specification languages OCL and JML do not rely on the *Black Box* used so far for the semantical foundation of Java.

We can get rid of that problem by changing the semantics of implementation models.

Java semantics in terms of logic There is quite a number of logical systems [11,4,12,1] which (partially) capture the semantics of Java[1] formally.

[1] Unfortunately, some advanced concepts of Java, e.g. threads, reflection, are still excluded.

Furthermore, each of the logical systems is supported by the tools Isabelle [5], Loop-Tool [9], LOPEX [10], KeY-System [7], respectively. The support by sophisticated tools makes it feasible to check mechanically certain properties of Java programs.

Suppose, the implementation model is semantically based on the logical system described in [1] (Dynamic Logic). Then, conformance tests are enabled against all those models which are formulated within a language *similar* to the language of Dynamic Logic. Two languages are called *similar* iff they are defined on comparable semantics. For instance, OCL is similar to Dynamic Logic, and JML is similar to the logical system given in [4], since there are implemented translations integrated in the KeY-System and the Loop-Tool, respectively.

The languages OCL and JML are much more suitable than Java to describe abstract properties. For the `addTwo`-example, the following OCL-constraint formalises the intended requirement:

```
context Foo::addTwo(int i)
post:     result > i
```

A conformance test between this model and the implementation model can be processed by the KeY-System fully automatically. Internally, the KeY-System manipulates symbols such as `i`, `result`, etc. Therefore, we name such conformance tests *symbolic conformance tests*.

Conclusion

The MDA advocates the description of a system as a collection of several models. However, the languages to formulate the models are left open. This paper argues that machine-based conformance tests are only feasible between those models, which rely semantically on comparable definitions. We propose a classification of conformance tests for implementation models.

Conformance tests against the implementation model can be classified based on the style of semantical description for the programming language.

In a first case, we considered the programming language be semantically given by a reference to its compiler. Then, the other model used in the conformance test must semantically be given by a reference to the same compiler. Thus, both models and the conformance test are executable.

In the second case, the programming language is semantically given in terms of logical systems. This enables conformance test between the implementation model and a second model written in an expressive specification language, e.g. OCL, JML. The conformance test itself can be carried out using tools such as the Loop-Tool or the KeY-System. Since all such tools work at a symbolic level, user interactions are sometimes required. However, user interaction can be reduced to a minimum by the usage of a mature tool.

Acknowledgements

My thanks are due to Martin Giese, Reiner Hähnle, and Bernhard Beckert for their comments on earlier drafts of this paper.

References

1. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001. 232, 233
2. D. D'Souza. Model-driven architecture and integration – opportunities and challenges, 2001. Available from `www.kinetium.com`. 231
3. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1997. 231
4. M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Proceedings, Fundamental Approaches to Software Engineering (FASE), Berlin, Germany*, LNCS 1783. Springer, 2000. 232, 233
5. Isabelle. *Generic Theorem Proving Environment. Web pages*. Cambridge University and TU Munich. Available from `www.cl.cam.ac.uk/Research/HVG/Isabelle`. 233
6. JUnit. *Web pages*. Available from `www.junit.org/index.htm`. 232
7. KeY Project. *Integrated Deductive Software Design. Web pages*. University of Karlsruhe and Chalmers University Gothenburg. Available from `i12www.ira.uka.de/~key`. 233
8. G. T. Leavens, K. R. M. Leino, C. Ruby, and B. Jacobs. JML: Notations and tools supporting detailed design in Java. In *OOPSLA'2000 Companion*. ACM, 2000. 232
9. Loop Project. *Logic of Object-Oriented Programming. Web pages*. Katholieke Universiteit Nijmegen. Available from `www.cs.kun.nl/~bart/LOOP`. 233
10. LOPEX Project. *Logic-based Programming Environments. Web pages*. FernUniversität Hagen. Available from `www.informatik.fernuni-hagen.de/import/pi5/forschung/lopex.html`. 233
11. D. v. Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS 1523. Springer, 1999. 232
12. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Proceedings, Programming Languages and Systems (ESOP), Amsterdam, The Netherlands*, LNCS 1576, pages 162–176. Springer, 1999. 232
13. Rational Software Corp. et al. *Unified Modelling Language Semantics, version 1.3*, June 1999. Available from `www.rational.com/uml/index.jtmpl`. 232
14. R. Soley. Model driven architecture, 2000. White paper. Available from `www.omg.org/mda`. 231

Object-Oriented Theories for Model Driven Architecture

Tony Clark[1], Andy Evans[2], and Robert France[3]

[1] King's College London, UK
anclark@dcs.kcl.ac.uk
[2] University of York, UK
andye@cs.york.ac.uk
[3] University of Colorado, USA
france@CS.colostate.EDU

Abstract This paper proposes a number of generic modelling technologies that can be used to support the OMG initiative for Model Driven Architecture (MDA). Object theories are used to combine these technologies into a meta-modelling framework for MDA.

1 Introduction

This paper reviews the requirements for the current OMG initiative for Model Driven Architecture (MDA) and proposes a number of generic modelling technologies that can be used to support MDA. MDA is concerned with modelling all aspects of the software development process. In particular, MDA distinguishes *platform independent models* (PIMs) from *platform specific models* (PSMs). PIMs capture the structure and behaviour of a system independent of any particular implementation platform. PSMs are system models expressed in terms of particular implementation platforms (e.g., .NET, EJBs, C++). A PIM may be mapped using MDA to many different PSMs. The aim of MDA is to decrease the amount of implementation detail required in system development by taking advantage of patterns of implementation. MDA also aims to raise the level of abstraction and integration at which system developers work. The OMG web site (http://www.omg.org) contains a number of documents that discuss the emerging features of this new field. Readers looking for an overview of MDA should start with [11] and [7].

In this paper we propose technologies that can be used to support MDA, including: package specialization that support a modular approach to MDA; templates that support a pattern driven approach to MDA; non-intrusive relations that support an abstract approach to modelling mappings; and, object theories that combine these technologies into a meta-modelling framework for MDA. Space does not allow us to describe these technologies in depth. For a more detailed account see [14].

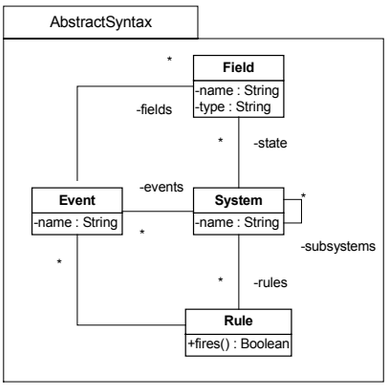
J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 235-244, 2002.

2 Language Definitions

In order to support MDA, languages must be precisely defined, that is, they must have precisely defined syntax and semantic domains and a well-defined relationship between these two domains. These three essential components can be modelled using fairly simple UML concepts. A precise semantics is essential for defining complete and sound relationships between PIMs and PSMs and between all other MDA components.

A precisely defined language consists of (1) a concrete syntax (the human-centric view of the language as it is typed into an editor, drawn on a screen or printed on a page); (2) an abstract syntax (the computer-centric representation of concrete syntax suitable for manipulation as data structures in a program); (3) semantic domain elements that define the denotation of syntactic elements; and (4) well-defined mappings that link concrete to abstract representations, and abstract representations to their meaning.

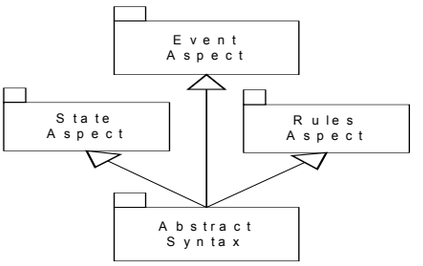
We will use a simple language to express key features of language engineering technology for MDA. The language is intended to support the system analyst when capturing business rules. A system definition consists of the following features: a name; a set of sub-systems; a set of named input events consisting of simple fields; a state consisting of a set of fields; and a collection of rules. Each rule has an antecedent that is a condition expressed in terms of the fields on an input event and a consequent that is a pair consisting of a collection of output events and a condition on the system state. The abstract syntax definition is given above (details on how rules are represented are omitted). A rule *r* has an operation "fires", where *r.fires(e,S,S')* is true when *e* is an event that triggers *r*, *S* is a set of fields defining the pre-state of the system and *S'* is a set of fields defining the corresponding post-state. A system executes by selecting a rule whose antecedent is enabled by some input event. The input event is consumed and the corresponding output events are produced along with the required change in system state. The abstract syntax model given above is a meta-model; its instances are models in the sense that a model for Java has instances that are programs; the programs then have instances that are computations. A *computation* for a business system is a history of system states where state transitions occur due to system events. A state in a computation is a set of named values. A computation is a well-formed meaning for a business system when the events in the computation match up to the events in the system model and the names used in system states match the fields given in the system model. In addition, each transition in the computation must fire at least one of the rules in the system model.



3 Compositionality

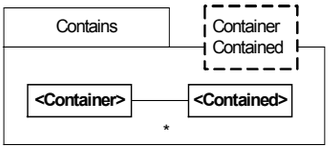
Compositionality of the MDA process is essential since the efficacy of the approach will depend on controlling the complexity of large numbers of models and their interrelationships. Compositionality within MDA can be achieved through the use of *package specialization*. A package is a container of model elements. If one package specializes another package then it includes all of the elements from the super-package and adds new elements of its own. Where the names of corresponding model elements overlap (due to local definitions or multiple inheritance) the elements are merged.

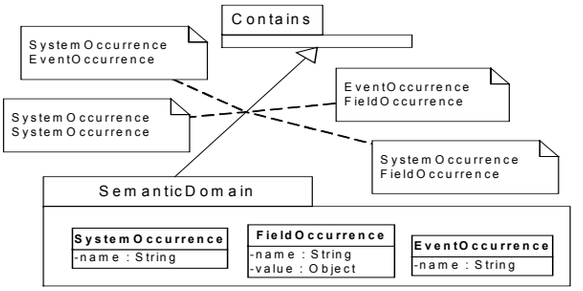
For example, suppose that we wish to decompose the definition of the business language abstract syntax into three aspects: the events that can be received from the environment; the rules that control the reaction of the business system; and, the state of the business. This can be achieved by defining each aspect in a different package. packages are then combined using package specialisation as shown on the left. Merging rules ensure that the different aspects are combined correctly to produce the required definition of abstract syntax. The package composition is shown on the left.



4 Patterns

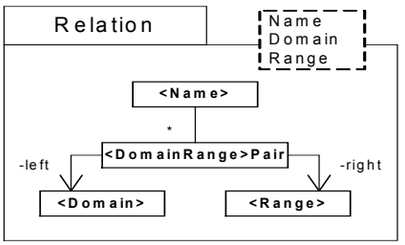
Patterns will arise in language definitions and MDA. For example, many languages exhibit standard properties such as containership and dynamic behaviour. The relationships between languages can also exhibit standard patterns in terms of structure or behaviour preservation. Patterns can be expressed using *template packages*. A template package is parametric with respect to model elements. A template is *stamped out* to produce a corresponding package in which all the formal parameters have been replaced with the actual parameter values. For example, containership is a pattern that occurs repeatedly in the abstract syntax of the business language. A template is expressed as a package whose parameters are listed in a dashed box on the top right. In the body of the package the parameters are referenced within chevrons. The actual parameter values are supplied in a box attached to an arrow linking the resulting package with the template. The same template may be stamped out multiple times in which case each occurrence is identified by a different parameter box. The structural features of the semantic domain for the business language stamped out using the containership template is shown below. The term *occurrence* signifies that the corresponding elements are execution time entities domain.





5 Relations

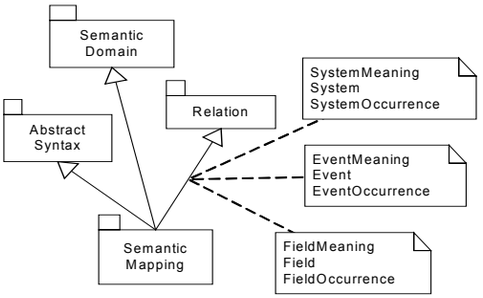
MDA relies heavily on relations and mappings. Relations should be non-intrusive (domain and range do not necessarily have to know about the relationships). Relations need to be compositional and there are many standard relationship patterns.



The template on the left can be used as the root of many relation patterns. The relation is named and has a domain class and a range class. The relation consists of a collection of pairs each containing an instance of the domain and an instance of the range. There is no restriction on the relation regarding which domain and range elements are paired. Standard relationship properties such as functional and one-to-one can be expressed as specializations of the basic relationship template. This leads to a powerful language for specifying relationships, see [7] for more details.

6 Semantic Mapping (System Meaning)

In general a semantic mapping will involve both structural and non-structural constraints that determine legal pairings of syntax and semantics elements. We would like the semantic mapping to exhibit the following features: it should be simple, easily understood and use diagrammatic notation where possible; it should be able to reuse patterns where possible; it should be compositional and



modular; where possible it should be amenable to machine processing since it can be used to generate test cases and even to animate a language definition

We propose a novel technique called object theories that allows mappings (including semantic mappings) to be expressed such that the criteria given above are achieved. The diagram on the left constructs a free semantic mapping in which no constraints limit the pairing of syntax and semantic elements; the constraints will be given as object theories as described in the following sections.

7 Object Theories

OCL is used to express well-formedness constraints and to define the set of pairs associated with a relation. Experience in defining the UML 2.0 RFP submission [13] would suggest that the combination of templates and OCL to express such constraints is powerful, but can become complex in the sense that some of the structure of the constraints is lost in the OCL language. Object theories are a mechanism that allows the constraints to be structured and composed in layers. This section introduces object theories and subsequent sections discuss a new mechanism for expressing the theories and combining them. Examples are given with respect to the simple business language.

An object theory is a collection of UML static semantics snapshots. A collaboration diagram or a sequence diagram can be viewed as the presentation of an object theory. Typically we define the structure of a domain by stamping out templates and combining the resulting packages using package specialization. This in itself defines an object theory; but typically one that expresses a free structure, i.e. there are no restrictions on the linkage between objects that are instances of the resulting classes. Such a freely constructed domain model is then specialized by combining new object theories that rule out certain object configurations. This process is incremental, modular and reversible.

A method for MDA using object theories proceeds as follows: all the domains are defined independently using template technology; object theories can be used to define well-formedness rules on the freely constructed domains; the structure of semantic mappings are defined using templates; object theories are then used to complete the constraints over the freely constructed semantic mappings; the resulting language definition is can then be related to other language definitions (perhaps PIM to PSM languages) by template defining freely constructed relations and then *mixing in* constraints using object theories. Object theories are constructed by combining partial theories. This raises an issue regarding consistency: are there any models of the resulting theory? This is an important issue and must be checked, ideally using tool support.

8 Object Theory Presentations

Object theories, as described above, are not particularly new since any UML diagram can be viewed as an object theory. However, UML notations provide only partial sup-

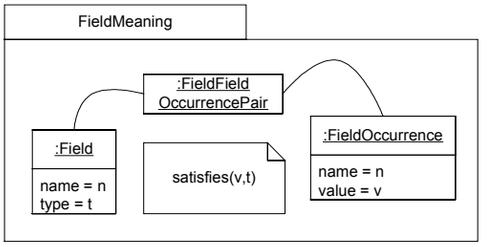
port for object theories and do not precisely define how such theories can be combined.

We propose a new type of diagram and equivalently a textual language for UML object theory presentation. An *object theory diagram* (OTD) is based on object diagrams together with OCL constraints over the variables used to name objects. The differences are as follows: OTDs are expressed in packages and are amenable to package specialization and template definition; OTDs can be partitioned into sub-OTDs understood to be combined using disjunction; OTDs can be defined as *axioms* or *rules* where an OTD axiom is just an object diagram + OCL and a rule has a number of antecedent object diagrams, a consequent object diagram and an OCL side condition; OTDs allow links to be expressed as *combination* links or singleton *links*. A combination link leads to an object representing the set of values linked to the object where the link labels all have the same name. A singleton link is one of the possibly many links with a given label leading from an object. Groups of links are labelled to determine whether they are partial (there may be other links with the same label that are not shown on the diagram) or total (all the links with the given label are accounted for).

A theory can also be expressed as a definition in a textual language. The language is more amenable to mechanical processing being easy to construct without sophisticated graphics based tools. A theory named T is expressed as a definition:

theory T [**extends** *theories*] *body* **end**

The body of the theory is a constraint on the valid theorems. A theory can be defined as an extension of an existing theory in which case all of the extended theorems are imported into the theory. The body of a theory is a disjunctive collection of rules. Each rule has an antecedent and a consequent (the degenerative case occurs where the antecedent is empty). The antecedent and consequent of rules are snapshots which are collections of object patterns. The textual language for object theories is more flexible than OTDs which can easily get cluttered. In addition, the textual language can easily be extended with new theory operators that might be difficult or clumsy to express on a diagram. In the rest of this paper we will use both notations.



The example on the left shows an OTD axiom defining the semantics of fields. A field occurrence is a valid meaning for a given field if the names agree and the value of the occurrence satisfies the type declared for the field. When this theory is combined with the free definition of the relation FieldMeaning the legal instances of FieldFieldOccurrencePair are constrained so that the names agree and the type of the value is legal.

The theory FieldMeaning gives a simple example of how theories can be used to construct relationships between models. In general a relationship can be defined as a class with associations to n (in the case or FieldMeaning n = 2) model elements. The theory defines when the relationship is well formed by placing constraints on the asso-

ciated elements. Complex relationships can be decomposed into combinations of simpler relationships.
The textual representation of the theory is as follows (from now on to save space long class names will be abbreviated):

```
theory FieldMeaning
  obj :FFOP
     left = obj :Field name = n; type = t end;
     right = obj :FO name = n; value = v end
  end
  such that satisfies(v,t)
end
```

Just as patterns occur in static structure (for example containment), patterns occur in object theories. Package templates are used to express patterns in static structure; object theory templates are used to express patterns arising in theories. Object theory templates are illustrated in [14].

9 Refactoring and Refinement

One of the key features of MDA is the ability to define relations *between* languages. These relations must hold between both the abstract syntax domains and the semantic domains of the languages. In order to be useful, the relations between languages must explicitly define the properties that they preserve between the languages.

Relations may hold between instances of the same language. The *refactoring* relationship is one such relationship. Refactoring occurs when a software artifact (models or code) is changed to improve specific qualities while preserving the essential properties of a system. A refactored artifact represents an improvement in how a desired result (characterized by the essential properties) is accomplished. This section gives a very simple example of a refactoring with respect to the example business language.

The business language supports sub-systems. The definition of execution for the language requires that sub-system steps are performed during a single super-system step. One possible refinement of a system is to decompose a single system into two sub-systems. If system A is refactored into A′ consisting of two sub-systems B and C then the events handled by A are also handled by A′, but instead of causing a state change in A, A′ delegates the event to either B or C. Therefore the events handled by both B and C are disjoint sub-sets of the events handled by both A and A′. The state of A′ is empty and the state of B and C are disjoint sub-sets of the state of A.

Given a set of events E, a set of fields F and a set of rules R suppose that complete(E,F,R) holds when all rules in R handle only events in E and require only fields F to determine whether they are enabled and to determine the result of firing the rule. Suppose that given a set of rules R, a set of events E and the name of a sub-system n that delegate(R,E,n) holds when the rules in R all handle events in E and delegate the event to the sub-system n. A simple binary refactoring is expressed by the following theory named BinSplit. The theory defines a relationship called Split between a system (left) and its decomposition into two sub-systems (right):

```
theory BinSplit
  obj :Split
    left =
      obj :System
        fields=F1->union(F2); events=E1->union(E2);
        rules=R1->union(R2)
      end;
    right =
      obj :System
        events=E1->union(E2); rules=R1'->union(R2');fields=Set{};
        subSystems=Set{
          obj :System name=n1; events=E1; rules=R1; fields=F1 end,
          obj :System name=n2; events=E2; rules=R2; fields=F2 end}
      end
    end
   such that complete(E1,F1,R1) and complete(E2,F2,R2) and
             delegate(R1',E1,n1) and delegate(R2',E2,n2)
end
```

A theory of refactoring must also define how semantics are restructured and then show that the transformations preserve certain properties. Since out approach uses modelling techniques to express both the syntax and the semantics of a language, we can use the same techniques to express a theory about semantic refactoring and use the theory to check the appropriate properties.

A systematic, automatable approach to model refactoring is possible when the transformations need to accomplish refactorings involving well-defined sets of source and target models, can be precisely characterized. An approach to UML model refactoring, based on the notion of UML (meta) Role Models has been developed [8]. A Role Model is a restriction of the UML meta-model, that is, a Role Model determines a sublanguage of the UML. One can view a Role Model as a characterization of models (precisely those models that can be expressed in the UML sublanguage). A particular type of Role Model that has been A particular type of Role Model that has been used to define refactorings is the Static Role Model (SRM). An SRM is a characterization of static UML models (e.g., Class Diagrams). In this approach, a characterization of a family of model refactorings consists of the following elements: A source model set: This set consists of source models for the refactorings. The set is characterized by Role Models; A target model set: This set consists of target models for the refactorings. The set is characterized by Role Models; A transformation set: This set consists of transformations that each accomplish the refactoring goal. A transformation specifies how a source model is transformed to a target model. In our work, transformations are expressed in terms of subtransformations a source model undergoes as it is transformed to a target model. An example of a pattern-based design refactoring is illustrated in [8].

10 Implementation

A key feature of MDA are mappings from PIMs to PSMs. In general implementation involves several inter-language relationships. Object theories can be used to construct such relationships. This section gives a simple example of an implementation.

Suppose that we have a model of a simple dynamic modelling language involving packages, classes, attributes and methods. A method has a collection of parameters, a body that is a simple action and a post-condition in terms of the attributes of the containing class. The semantic domain of this language has definitions for objects and object steps. Each object step is labelled with the method call that caused it to occur.

This language can be used to implement the business language by using object theories to construct the following relations: state fields become attributes; events become method signatures; rule conditions become the pre and post-conditions of the method that corresponds to the event that triggers the rule; rule actions become method bodies; systems and sub-systems become classes; sub-systems of systems are flattened. Space limitations prohibit a detailed description of the transformations. See [14] for more details.

11 Conclusion

In this paper we identify requirements for an MDA approach to system development and outline a framework that consists of principles, techniques and mechanisms that support an MDA approach. The framework is intended to guide our work on developing an integrated set of processes, techniques, and mechanisms that cover all aspects of system development and that raises the abstraction level at which system engineers develop implementations above the code level.

The approach to language engineering has been developed as part of a submission to the UML 2.0 revision initiative. Further details about package specialization can be found in [4], examples of the use of templates to construct a modelling language can be found in [3] and a complete definition of the UML core infrastructure can be found in the current submission to the OMG [13]. The initial ideas related to package specialization and templates are drawn from Catalysis [6]. Information about defining and combining aspects of systems can be found in [5].

This work is related to other research that addresses expressing constraints and relationships between models such as [9], [12] and [1]; and in particular with proposals for graphical constraints. The constraint diagrams of [10] and the graphical OCL diagrams of [2] are clearly related to this work. However, any technology that supports MDA must allow relationships between models to be *performed* in the sense that given a complete (or partial) description of one side of a binary relationship there should be an effective procedure that calculates the other side of the relationship. Our work is novel in that it aims to achieve this by restricting the form of *theories* that define the relationships in the same way that Horn Clauses restrict the form of logical formulas.

We intend to develop the language of object theories and to define a complete semantics for the language (using the techniques defined in this paper). In addition, we intend to build a tool that can perform object theories and to use this to develop a theory of MDA based on definitions of model transformations such as refactoring and refinement.

References

1. Akehurst D. H. and Kent S. (2002) A Relational Approach to Defining Transformations in an Metamodel. UML 2002, Dresden.
2. Buttoni P. et al. (2001) A Visualization of OCL Using Collaborations. In Gogolla and Kobryn eds. UML 2001 - Modeling Languages Concepts and Tools. Toronto, Canada 2001.
3. Clark A., Evans A., Kent S. (2002) Engineering Modelling Languages: A Precise Meta-Modelling Approach. Presented at the ETAPS FASE Conference, Grenoble France, 2002.
4. Clark A. Evans. A. Kent S. (2002) Package Extension with Renaming. To be presented at UML 2002, Dresden, Germany.
5. Clarke S., Walker R. J. (2001) Composition Patterns: An Approach to Designing Reusable Aspects, in Proceedings of ICSE 2001, May 2001.
6. D'Souza D., Wills A. C. (1998) Object Components and Frameworks with UML – The Catalysis Approach. Addison-Wesley.
7. D'Souza D. (2001) Model Driven Architecture and Integration. Available from http://www.catalysis.org/omg/.
8. France R., Kim D., and Song E. (2002) Patterns as Precise Characterizations of Designs. Technical Report 02-101, Computer Science Department, Colorado State University, January, 2002.
9. Gogolla M., Richters M (2002) Expressing UML Class Diagram Properties with OCL. In Clark A, Warmer J. (eds) Advances in Object Modeling with the OCL. Springer Verlag, LNCS 2263.
10. Kent S. (1997) Constraint Diagrams: Visualizing Invariants in Object-Oriented Models. In Proceedings of OOPSLA '97, 327 -- 341.
11. The OMG (2001) Executive Overview: Model Driven Architecture. Available from http://www.omg.org/mda/.
12. Richters M., Gogolla M. (2000) Validating UML Models and OCL Constraints. In Evans A., Kent S., Selic B. (eds) UML 2000 The Unified Modeling Language – Advancing the Standard. Third International Conference. York, UK 2000. Proceedings volume 1939 LNCS, 265 -- 277, Springer-Verlag.
13. 2UP UML 2.0 (2002) Infrastructure Submission. Available from the OMG web site at: http://www.omg.org/techprocess/meetings/schedule/UML_2.0_Infrastructure_RFP.html.
14. Object Theories White Paper (Clarke, Evans, France). Available at: http://www.cs.colostate.edu/~france/theories.pdf.

Systems Engineering Foundations of Software Systems Integration

Peter Denno and Allison Barnard Feeney

National Institute of Standards and Technology
Gaithersburg, MD 20899, USA
peter.denno@nist.gov
abf@nist.gov

Abstract. This paper considers systems engineering processes for software systems integration. Systems engineering processes, as intended here, concern how engineering capability should be factored into problem-solving agencies for performance of software systems integration tasks. Systems engineering processes also concern how the results produced by these agencies should be communicated and integrated into a system solution. The environment in which systems integration takes place is assumed to be model-driven. In the proposed solution, problem-solving agencies, working from various viewpoints, employ differing notations and analytical skills. In the course of identifying the systems engineering process, the paper presents a conceptual model of systems engineering, and reviews a classification of impediments to software systems integration.

1 Introduction

Software systems integration entails systems engineering, whether one consciously practices it or not. Systems integration starts with the recognition of new requirements and is not complete until one validates the resulting system against those requirements. A premise of this paper is that the choice of a systems engineering process is a matter of significant concern, particularly if one hopes to automate portions of the process in a model-based environment. [1]

A *model-based environment* (MBE) is an environment for systems integration that emphasizes the role of models in automating an integration process. The environment envisaged in this paper would provide an incrementally refinable account of a business process and the system that implements it. The account is derived from the union of all available views of the system. Views are provided in various notations (or "viewpoint technologies"). The MBE serves three purposes: (1) it fosters coherence among views by recognizing refinements and

[1] Commercial equipment and materials are identified in order to describe certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 245–259, 2002.

interrelations among disparate models; (2) it enables communication between problem solvers (human or automated) working in differing viewpoints toward the resolution of an integration task; and (3) it provides links from views that state or elaborate requirements to those that posit design commitments.

The concerns just mentioned — heterogeneous views, viewpoint communication, requirements and their allocation[2] to problem solvers — beg the question of what sort of systems engineering process the MBE should implement.[3] The answer to this question is the central point of this paper. The issue (1) above, of providing inter-model coherence is discussed in [8].

Section 2 of the paper describes a conceptual model of systems engineering. *Section 3* summarizes previous work which presents a classification of impediments to the integration of software-intensive systems [3]. *Section 4* concerns the central point of the paper. It considers choices of systems engineering processes for an MBE for software systems integration. *Section 5* outlines our MBE architecture. *Section 6* discusses related work and future plans for the MBE described in the paper. *Section 7* provides conclusions. The final section, *Appendix A*, is a glossary of terms from the systems engineering conceptual model.

2 A Conceptual Model of Systems Engineering

Systems engineering is any methodical approach to the synthesis of an entire system that (1) defines views of that system that help elaborate requirements, and (2) manages the relationship of requirements to performance measures, constraints, components, and discipline-specific system views. *Figure 1* provides a Unified Modeling Language (UML) class diagram of a conceptual model of systems engineering. *Appendix A* provides definitions of concepts presented in the UML.

The key terms in this definition of systems engineering are "requirements" and "views." Systems engineering is foremost about stating the problem, whereas other engineering disciplines are about solving it. A *view* is a representation of the whole system from the perspective of a related set of concerns [14]. An example of a view that "help[s] elaborate requirements" is a functional model of the system. It is a view that identifies the resources, roles, and processes involved in fulfilling the purpose of the system. A *viewpoint* is a method founded on a body of knowledge of some engineering or analytical discipline and used in constructing a view (derived from [14]). A view is an application of a viewpoint.

[2] The term *requirements allocation* refers to the task of charging problem solvers with the task of meeting requirements. *Technical problem solvers* meet requirements by making design commitments. *Conceptual problem solvers* refine original requirements and assert derived requirements.

[3] We distinguish a *systems engineering process* from a *design methodology* in that a design methodology only prescribes a path through the space of possible refinements (or design commitments). A systems engineering process prescribes the decomposition into agencies (viewpoints, problem solvers) that make the commitments, and a means to orchestrate these agencies.

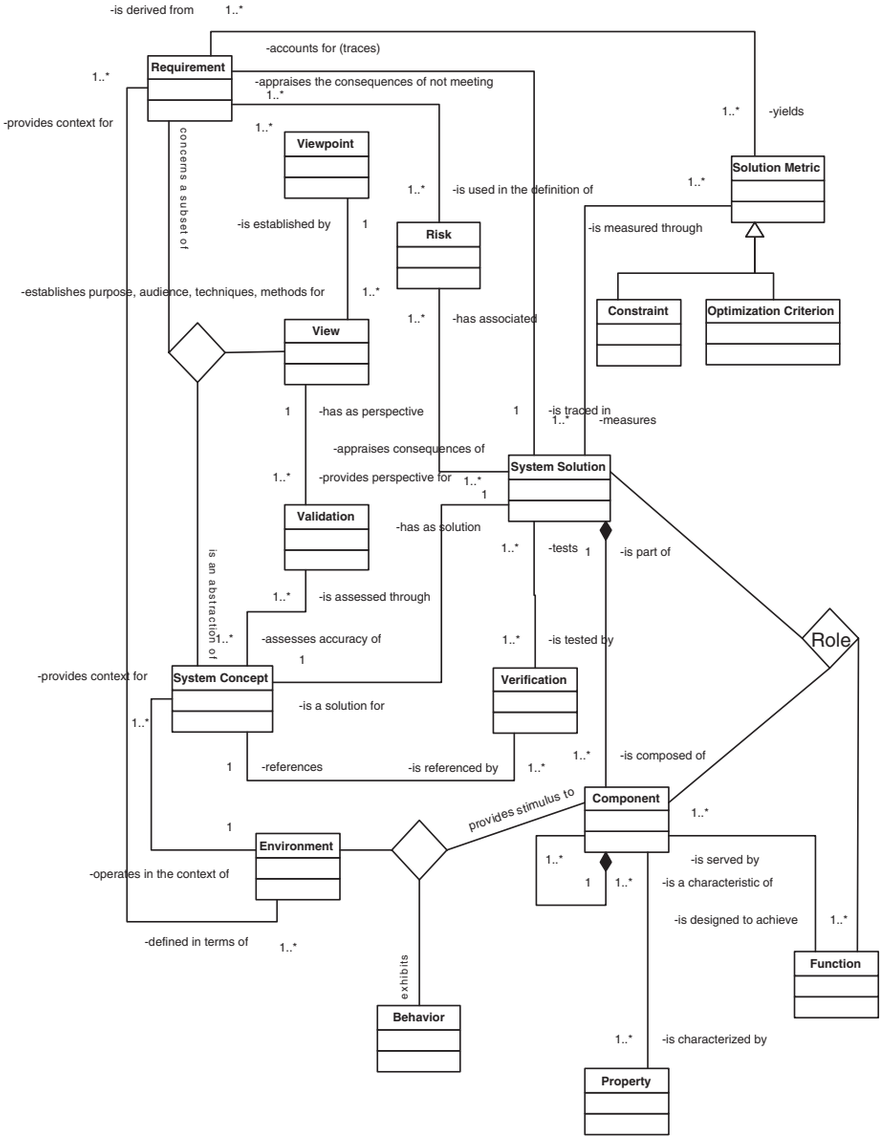


Fig. 1. Conceptual model of systems engineering

An architecture description language (ADL), such as the Wright language [1], provides a viewpoint that serves this role in systems engineering. Because an ADL describes a functional model of the system,[4] it posits, to some level of specificity, a componentization of the system and an implied correspondence to requirements. That is, every component has its purpose and that purpose can be traced to the requirements.

A view represents a collection of requirements. The utility of a view depends largely on whether or not there exists a corresponding body of knowledge and technology to draw from, that is, whether its viewpoint exists. The body of knowledge underlying the viewpoint (the technology and expertise in its use) may serve to refine requirements, assess how well a design meets the requirements, or it may posit a design or design commitments that would satisfy those requirements [17].

2.1 Requirements, System Solution, and System Concept

Requirements should make reference to the environment in which the system will operate. Requirements are about the environment [30]. The environment includes those components of the original system that remain unchanged (in structure and usage) in the new system. The environment also includes components whose usage is mandated. A *system solution* is an assembly of interacting components forming a whole and serving requirements. A system solution is an instance corresponding to its system concept. The *system concept* is a conceptual entity; it is a characterization, based on the requirements, that holds for anything that satisfies the requirements.

In engineering design and its formalization and automation, it is important that statements of requirement do not presuppose mechanisms intended to achieve those requirements. An MBE should distinguish viewpoints and problem solvers that make design commitments from those that refine statements of requirements and produce derived requirements without making design commitments. The reasons for this stipulation concern the invariance of original requirements and the relative tractability of a design process that relies on this fact. The following example illustrates this point.

Suppose that one has defined the original requirements of a new business process. A problem solver with an enterprise view may identify the components that are involved in fulfilling these requirements by identifying the relevant information, who possesses it, and who requires it. This problem solver may posit two derived requirements: (1) a derived requirement stating that an information flow must occur between the identified components, and (2) a derived requirement stating the triggering and temporal ordering of events that lead to the exchange

[4] In models that embody commitments to mechanism, *function* is defined as the activity by which something fulfills it purpose. However, in models that do not make commitments to mechanisms (*i.e.* conceptual models), this notion of function cannot be represented. *Utility* – what purpose something serves – is the the corresponding term in conceptual models.

of the information. These new assertions draw on knowledge of the environment (of both the existing and to-be system). Other than to identify roles, that knowledge does not concern the means by which the business process is achieved. As long as the original requirements remains valid, the derived requirements need never be retracted. The assertions made by these problem solvers are part of the system concept.

On the other hand, any of a number of problems might impede the implementation of this information flow (see *section 3*). If the problem is that both components behave as servers, one must take initiative to update the state of the component requiring the information. Potential solutions include: (1) component A, who has the information, triggers a process to communicate with component B, who needs the information; (2) a delegate of component B polls component A for changes in the information; or (3) both components A and B subscribe to an event service that is tracking the original production of the information. The choices made here are design commitments, since they concern mechanism, and their quality is contingent on design commitments made elsewhere to address other requirements. If, for example, the totality of requirements suggests that a new event-based channel of communication is warranted, choice (3) may become increasingly attractive. That is, it may become necessary to retract assertions that are design commitments. The assertions made by these problem solvers are part of a system solution, not the system concept.

2.2 Behavior, Function, and Role

Systems[5] are made of components, which themselves may be viewed as systems (subsystems to the system). Systems may be described efficiently in terms of their function, but in *ad hoc integration*,[6] when a subsystem is integrated into a system, what is most relevant is not what its function was as it operated in isolation, but rather whether its behavior serves a function in the planned system. This is so because it is ultimately the behavior of the subsystem that must be controlled and exploited to service the needs of the system. The function of the component as it was in isolation may not be relevant. For example, a text editor may be used to write reports. The editor may serve this function at various points in the processes of an organization. It may be the case that the editor has a regular expression search capability that can be used to to recognize exceptional conditions in data collected from the factory floor. The regular expression search behavior can be harnessed for this purpose. Its report writing function is not relevant here.

The behavior of a component may serve multiple functions, and each instance of its application in a system may serve a different function. The function of the

[5] Unless it is necessary to distinguish the concept of the system from its solution, we will henceforth use the term *system* to refer to a system solution. This is consistent with common usage.

[6] *ad hoc integration* is integration where a communication flow is required between components that were not conceived with the intention of providing or receiving that flow.

instance in the context of the system is called the *role* of the component. The notion of role cannot be distinguished from the notion of function, except for the fact that a role is defined in the context of the usage of an occurrence of the component in a particular system context.

3 Impediments to Integration

Integration is about enabling components to act jointly toward a goal. In the scope of the model-based environment, the problem solvers that are orchestrated in the systems engineering process work toward enabling joint action. Joint action is impeded by various obstacles. [3] identifies five broad categories of impediments to the integration of software-intensive systems and classifies problems within these. That work is summarized below.

Technical impediments concern problems in communication and process flow arising from the technology and logistics employed at the interfaces of components. These include control conflicts (*e.g.*, every component is designed to be a client, or every component is designed to be a server) and differences in the syntax of messages.

Semantic impediments concern how well information communicated among the components of the system serves the joint action that fulfills the purpose of the system. *Semantics* refers either to a theory of behavior or a theory of reference [25]. Terms have a sense and a reference. Regarding reference, communicating agents may differ with respect to the objects to which a term refers. These differences may be (but are not always) detrimental to joint action. Regarding sense, the behavior that a message elicits from the recipient may be in conflict with what is expected and intended by the speaker.

Information may be conveyed directly to known recipients or published. Information conveyed to known recipients intends to elicit particular behaviors from the recipients and utterances are designed for that specific purpose. Published information is information that should be true in the context of the system. The behavior that published information is intended to elicit is known only by the systems engineer, not the components that provide the information.

Functional impediments concern conflicts arising from a mismatch between actual behavior and the behavior that is expected for a particular role. An agent may perform activities that are beyond those called for in its designated role. The effect of these extraneous activities may be (but are not always) detrimental to joint action.

Qualitative impediments concern how well a component performs the role to which it is tasked. Qualitative impediments include accuracy, security, trust, credibility, and timeliness of results.

Logistical impediments concern the impact of the designed system on the system in which it is embedded. Problems here concern system validation, that is, how well the deployed system, in fact, satisfies requirements. Important requirements may have gone unstated and unfulfilled. Requirements may have changed while the system was being designed.

4 Systems Engineering Processes

A *systems engineering process* prescribes a decomposition of engineering capability into problem-solving agencies, and a means to orchestrate these problem solvers. A design methodology prescribes a path through the space of potential refinements (of both requirements and specifications). Differing design methodologies can be employed within the individual problem solvers as the means by which they function. The choice of systems engineering process determines how requirements are classified and allocated to problem solvers. In part because different system engineering processes suit different problems, systems engineering has not defined a best-practice systems engineering process. Such a thing might not exist [11].

The engineering of complex systems typically follows a top-down and then bottom-up development process, depicted as a *V*, where the left half of the *V* represents the requirements definition and decomposition effort, and the right half of the *V* represents integration and verification [12]. (See *figure 2 (a)*). This basic flow holds generally, including *ad hoc* integration and business process reengineering situations. In more complex systems development, the conceptual architecture itself may be subject to modification during development. This process has been described as a *modified-V* pattern [28]. (See *figure 2 (b)*). This flow somewhat resembles one in software systems engineering in which one evolutionary path of development (and the "stove-piping" it entailed) is terminated and a reconceptualization is made to simplify broad areas of the design.

We are motivated in this discussion to identify the systems engineering process and problem-solving architecture that best address the nature and requirements of software systems integration in an MBE. These requirements include (1) the facilitation of heterogeneous viewpoints and notations that express the many relevant views of the system, (2) the communication of results among these viewpoints [30], (3) modularity that accommodates new viewpoints, (4) simplicity, (5) end-to-end (requirement definition to validation) completeness, and (6) leverage of our knowledge of the problems of systems integration, including classification of integration impediments.

Of these requirements, (1) and (2) appear to be most difficult to satisfy. Among the many issues here, knowing what information need be communicated among viewpoints and what form that communication should take are problematic. Agencies do not act in total isolation, however, the complexity of engineering the entire system requires that details that are inconsequential to the decisions of other agencies remain isolated in the agency from which these details originate [18]. Deciding what information needs to be exposed re-

quires systemic domain knowledge. An analogy from the systems engineering of hardware-intensive systems [22] illustrates the problem: Suppose a spacecraft project has a weight budget and a choice of integrated circuit (IC) technology must be made. It may appear that the choice has little to do with weight (all chip weights are approximately equal). However, one IC technology may entail greater power consumption and thus the need for additional solar power collectors and batteries. An analysis of similar problems, termed *nearly decomposable problems*, is provided by Simon [26].

We consider three general processes:

Discipline-centric process: A discipline-centric systems engineering process is one in which engineering capability is organized by engineering discipline (*e.g.*, network analysis, concurrency analysis). That is, engineering capability is organized into problem-solving agencies by classification of their area of expertise. Examples of this approach include the time-tested means by which many hardware-intensive systems such as automobiles, aircraft, and spacecraft are developed. The process is most effective when applied to the development of a class of similar products, where knowledge of the flow of information among agencies (that is, a design procedure) has been well established.

In this process, problem solving follows an established route.

Component class-centric process: A component class-centric systems engineering process is one in which engineering capability is organized by expertise in the development of a class of subsystems (*e.g.*, database, or factory floor data collection). This class is related to the discipline-centric process in that disciplines and subsystems tend to correspond. The process is most effective when applied to projects that require the development of new subsystems. The procedure is at a disadvantage in situations where the specification of the interfaces between components are subject to modification.

In this process, problem solving follows a route determined by functional decomposition.

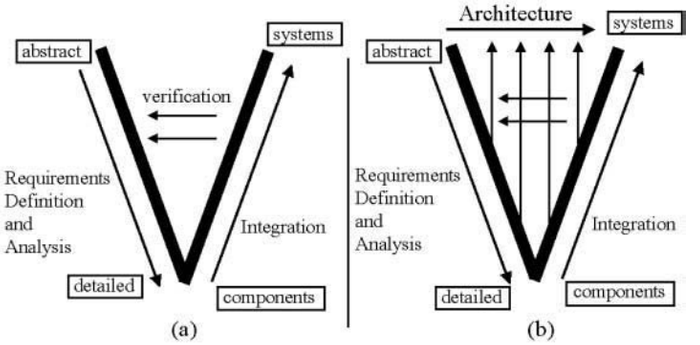


Fig. 2. Two patterns of systems engineering process

Problem-centric process: A problem-centric systems engineering process is a variation of a discipline-centric process in which engineering capability is organized to address classes of well-known problems. However, here no design procedure has been established. Instead, derived conceptual requirements identify instances of the well-known integration problems. The approach may become excessively complex if these problems are revealed not by derived requirements, but only through detailed design commitments.

In this process, problem solving follows a route that emerges only with refinement of requirements and assertion of design commitments.

Of the approaches considered, the problem-centric process appears to be most appropriate to the design of an MBE for software systems integration. A discipline-centric process is not responsive to the *ad hoc* emergence of integration tasks, as might be required to resolve a control conflict (see *section 3*), for example. A component class-centric process may avoid this problem but may be inappropriate where the concerns to be resolved are predominantly impediments to integration, not design syntheses. In the use envisaged, enterprise views of the baseline system provide a conceptual model upon which derived conceptual requirements can be formulated. Derived requirements that concern needed information flows, temporal ordering, and timeliness constraints between identified components provide a basis to review the subject components with respect to the impediments to integration (*Section 3*). The demarcation between requirements (invariantly asserted) and design commitments (subject to possible retraction) is the point at which the information flow, its temporal constraints, and a neutral representation of the information to be exchanged[7] are identified and provided to problem solvers corresponding to particular concerns. Design commitments made by problem solvers may introduce new integration concerns.

5 A Model-Based Environment for Software Systems Integration

This section describes an MBE for software systems integration based on the problem-centric systems engineering process. The approach, as suggested in the previous section, involves multi-disciplinary knowledge, heuristic techniques, and a flexible problem-solving ability.

Success in the use of the approach requires, foremost, a comprehensive collection of information, including enterprise models, information models, and technical models. The scope of information required must be sufficient to reveal obstacles to the implementation of the new business process. Some of the information necessary may exist as artifacts of earlier design efforts. Information such as a mapping of information structures to an ontology of concepts in the business process imposes an additional cost on systems integration. However, in

[7] This representation may be, for example, an ontology of concepts relevant to the business process being implemented. Problem solvers may need access to a view that maps these concepts to information structures they expose in interfaces.

the system envisaged, the ontology and references between models should provide enduring value; it can evolve with the enterprise system it represents. As described in [8], the information collectively embodies an enterprise model of the subject business processes. The model is an *emergent enterprise model* in the sense that it comes into being through the accretion and interrelation of the various models generated in the course of the development and evolution of the enterprise's infrastructure.

Two other obstacles, these concerning the 'human element' must be addressed: (1) original statements of requirement are typically informally specified, potentially inconsistent or wrong; and (2) modeling notation (especially graphical ones) may be interpreted in differing and potentially conflicting ways.

With regards to (1), implementation of the approach imposes the additional work of translating the original requirements into a formal notation consistent with concepts in the ontology of the business process. The consequences of (2) are that inferences made from such models are apt to be invalid. The system should recognize certain modes of use (*e.g.*, a UML class diagram used to represent conceptual entities versus a UML class diagram used to implement classes) and make inferences accordingly.

Figure 3 illustrates the basic concept of the envisaged system. The design uses a blackboard architecture [5] to orchestrate viewpoint-specific problem solvers. The blackboard tracks the developing system concept and system solutions. Technical problem solvers provide results that are contingent upon assumptions about prevailing designs. The systems engineering executor is responsible for partitioning design commitments by the assumption set on which they are based. An assumption-based truth maintenance system (ATMS) [6], [7] may serve this purpose. Conceptual problem solvers provide the systems engineering executor with derived requirements, which are not assumption-based.

The nature of the language on the blackboard is a matter of great importance in the design of such a system. The language is the means by which problem solvers communicate with each other. It must be capable of expressing requirements and specifications at varying levels of detail. Further, it must be capable of providing content to the various viewpoints of problem solvers. In designing a language with these capabilities there is a risk of disproportionate reliance on the representation scheme of some given problem solver. To reduce this risk, the systems engineering executor component must operate only on information that is meta-level to the content that motivates the problem solvers. This requires that the language distinguish (1) problem domain content from (2) constructs that concern the structure and modality of the content, and provide directives to and from the executor. The nature of the designed systems engineering process is affected primarily by choice of representation scheme for message structure, modality, and executor directives, whereas the nature of the designed domain problem solving capability is affected primarily by the choice of representation scheme for the content language.

The purpose of the content language on the blackboard is to present problems for solution by the problem solvers. Problem solvers are responsible for transla-

tion to and from the blackboard content language to whatever form they find useful. The language concerns requirements, specifications, and the refinement of both. Earlier work in this area has addressed issues of accommodating various notations [16], [31], and the refinement of requirements [30], [27]. More recently, interest in the *family of languages* concept of UML [4] has directed attention to providing a semantics and formal foundation to relate a collection of modeling notations.

The blackboard content language of the subject model-based environment may be patterned after the predicate logic-based representation described by Zave and Jackson [31]. It should accommodate a notion of refinement such as that found in the Z specification language [27].

Problem solvers draw on *prescriptive views* (views corresponding to the problem-solving tasks for which they are specialized) as well as *concept links* that resolve terms referenced in requirements to information structures relevant to the views on which they work. Prescriptive views and concept links (views themselves) are retained in a model repository based on the OMG meta-object facility [20]. Terms referenced in requirements correspond to concepts defined in the enterprise ontology. The enterprise ontology could be implemented in a description logic such as Powerloom [24].

Prescriptive views are typically founded on existing viewpoint technology. Although a complete analysis of the relationship between prescriptive views and the integration impediments on which they have bearing is beyond the scope of this paper, the following observations can be made:

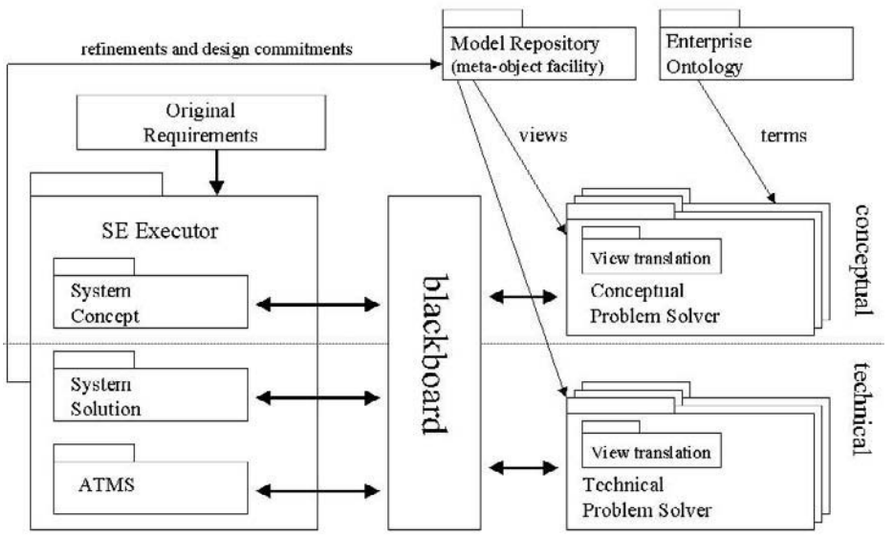


Fig. 3. System design

- Prescriptive views serving technical impediments include ADLs [1] and behavioral modeling notations such as statecharts [13].
- Prescriptive views serving semantic impediments include database schema, ontologies, and tools that identify conceptual differences in these.
- Prescriptive view serving functional impediments include the functional specification of components, activity diagrams such as IDEF0 [10], statecharts [13], functional flow block diagrams [23], and enterprise models [9].

Finally, the following observations can be made about this design:

- The design does not preclude the possibility that some problem solvers could be human.
- The design does not entail the definition of a formal notation for prescriptive views that lack one (*e.g.*, statecharts). How a problem is resolved through use of the view is a choice left to the problem-solver implementation.
- The systems engineering executor (serving the role of initiator of the blackboard) does not require global knowledge concerning the expertise of problem-solvers.
- The systems engineering executor is not responsible for transformations to problem solver notations.
- Subject system design choices can be traced to requirements.

6 Related Work

The Model Driven Architecture (MDA) of the Object Management Group (OMG) describes an approach that may allow a model specifying system functionality to be realized on multiple platforms through auxiliary mapping standards, or through point mapping to specific platforms. The model, called a *platform independent model* (PIM), provides a specification of the structure and function of a system that abstracts away technical detail [19].

The MDA is primarily concerned with providing freedom in the selection of middleware technology, and hence addresses a significantly narrower class of problems than the approach suggested by this paper. The MDA approach assumes agreement among communicating components on four of the five integration impediments described in *Section 3* (functional, semantic, qualitative, and logistical), and resolves only a subset of technical impediments.

The MDA PIM provides the viewpoints provided by the UML. These concern structural and control commitments that can be directly transformed into interface specifications. The PIM does not address most semantic concerns, such as mismatch of scope, granularity of abstraction, and temporal basis [29]. A PIM may provide a UML activity diagram that, like *Z*, does not indicate the agents performing the activities. This can be an advantage when the matter of who should act as server is a design issue. However, as this paper suggests, the choice can be contingent upon a wide spectrum of related concerns.

UML Version 2 (currently under development) intends to provide a common underlying semantics for the viewpoints of UML [21]. This would further the

goals of MDA, and provide to this project valuable input towards developing the content language of problem solvers.

ARIES [16], and the work of Zave and Jackson [31], is related to this work in providing environments for representing requirements and refinement amid multiple viewpoints. Those works sought a common underlying semantics for composition of specification across viewpoints. Like UML Version 2, those works may serve to provide a content language for the problem solvers of our work. They differ from what we propose in that we use the blackboard executor to focus attention on the impediments to integration, and use a model repository for accruing inter-model dependencies.

7 Conclusion

In this paper we reviewed ideas from systems engineering so as to identify the full scope of challenges affecting the ability to automate tasks of software systems integration. We identified a systems engineering process based on the resolution of integration impediments as the most efficient towards meeting these challenges. The solution outlined implements the systems engineering process as a blackboard executor mediating problem solvers working from multiple viewpoints.

The resolution of certain integration problems involves design choices, and hence is likely to be addressed only through heuristic, knowledge-based problem solvers. Some semantic impediments resist solution through automated means. In these situations, it might be possible to create and rely on semantic links between information structures exposed at interfaces and a domain ontology. But this approach is untried and could prove costly.

Our implementation of the ideas presented in this paper has been limited to the development of a basic meta-object facility, components of a blackboard and investigation of problem-solver technology such as the description logic systems [24]. Work towards the system envisaged is continuing.

A Glossary of the Terms from the Conceptual Model

Behavior - how something acts in response to various stimuli
Constraint - an expression derived from a requirement that partitions system solutions into those that meet the requirement and those that do not
Environment - the context in which a system operates
Function - mode of action or activity by which a thing fulfills its purpose
Optimization Criterion - a mathematical expression derived from a requirement that provides an ordering and metric on system solutions indicating how well each solution meets the requirement
Requirement - an optative statement intending to characterize and identify a system solution
Risk - a probabilistic expression that appraises the consequences of not meeting particular requirements

Role - the characteristic function or expected function of a thing in the context of a system solution

Solution Metric - expressions derived from requirements that are used to measure system solutions

System Concept - the concept of an assembly of interacting components forming a whole and serving requirements

System Solution - an assembly of interacting components designed to meet requirements

Trace - an account of the relationship between a requirement and a design decision

Validation - the process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model [2]

Verification - the process of determining that a model implementation accurately represents the developer's conceptual description of the model and the solution to the model [2]

View - a representation of a whole system from the perspective of a related set of requirements (derived from the definition of the term in [14])

Viewpoint - methods founded on the body of knowledge of some engineering or analytical discipline and used in constructing a view. *N.B.*, viewpoints establish the purpose and audience of a view.

References

1. Allen, R., Garlan, D., "A Formal Basis for Architectural Connection," *ACM Transactions on Software Engineering and Methodology*. July, 1997. 248, 256
2. American Institute of Aeronautics and Astronautics, *AIAA Guide for the Verification and Validation of Computational Fluid Dynamics Simulations (G-077-1998)*, AIAA Standards Series, 1998. 258
3. Barkmeyer, E. J., (Editor). *Concepts for Automating Systems Integration*, NIST Interagency Report, National Institute of Standards and Technology, Gaithersburg, Maryland, To be published. 246, 250
4. Cook, S., "The UML Family: Profiles, Prefaces and Packages" *UML 2000 — The Unified Modeling Language: Advancing the Standard*, Proceedings of the Third International Conference, York, UK, Springer Lecture Notes in Computer Science, Vol 1939, October 2000. 255
5. Craig, I. D., *Formal Specification of Advanced AI Architectures*, Ellis Horwood Limited, Chichester, West Sussex, 1991. 254
6. de Kleer, J., "An Assumption-Based TMS," *Artificial Intelligence* 28(2): 127-162, 1986. 254
7. de Kleer, J., "A Perspective on Assumption-Based Truth Maintenance," *Artificial Intelligence* 59(1-2): 63-67, 1993. 254
8. Denno, P., Flater, D., Gruninger, M., "Modeling Technology for a Model-Intensive Enterprise," Proceedings of SSGRR-2001, *Infrastructure for e-Business, e-Education, e-Science, and e-Medicine*, Scuola Superiore G. Reiss Romoli, L'Aquila, Italy, July, 2001. 246, 254
9. Esprit Consortium AMICE (editors), *CIMOSA: Open System Architecture for CIM*, 2nd revised and extended edition, Springer-Verlag, Berlin, 1993. 256

10. Federal Information Processing Standards, *Integration definition for function modeling (IDEF0)*, National Institute of Standards and Technology, Gaithersburg, Maryland, 1993. 256
11. Gabb, A., "Requirements Categorization," *Requirements Working Group of the International Council on Systems Engineering (INCOSE)*, 2002. 251
12. Grady, J. O., *System Integration*, CRC Press, Boca Raton, Florida, 1994. 251
13. Harel, D., "Statecharts: a visual formalism for complex systems". *Science of Computer Programming* 8, 3, June 1987. 256
14. IEEE 1471, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. Institute of Electrical and Electronics Engineers, Inc., September, 2000. 246, 258
15. ISO/IEC IS 10746, ITU-T X.900, *Open Distributed Processing – Reference Model*, International Organization for Standards, 1996.
16. Johnson, W. L., Feather, M. S., and Harris, D. R., "Representation and Presentation of Requirements Knowledge," *IEEE Transactions on Software Engineering*, Vol 18, No. 10. October, 1992. 255, 257
17. Mark, W., et al., "Commitment-Based Software Development," *IEEE Transactions on Software Engineering*, Vol 18, No. 10. October, 1992. 248
18. Minsky, M., *The Society of Mind*, Simon & Schuster, New York, New York, 1985. 251
19. Object Management Group, *Model Driven Architecture (MDA)*, http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01, July 1, 2001. 256
20. Object Management Group, *Meta Object Facility (MOF) Specification*, Version 1.3: ftp://ftp.omg.org/pub/docs/ formal/00-04-03 March, 2000. 255
21. Object Management Group, *Request For Proposal: UML 2.0 Infrastructure*, ftp://ftp.omg.org/pub/docs/ad/2000-09-01, September, 2000. 256
22. Oliver, D. O., *Personal communications*. 252
23. Oliver, D. O., Kelliher, T. P., and Keegan, G. J., *Engineering Complex Systems With Models and Objects*, McGraw Hill Text, 1997. 256
24. Preece, A., et al. "Better Knowledge Management through Knowledge Engineering," *IEEE Intelligent Systems* 16:1, Jan-Feb, 2001. 255, 257
25. Quine, W. V., *From a Logical Point of View*, second edition, Harvard University Press, Cambridge Massachusetts, 1980. 250
26. Simon, H. A., *The Sciences of the Artificial*, Third Edition, The MIT Press, Cambridge, Massachusetts, 1996. 252
27. Spivey, J. M., *The Z Notation: A Reference Manual*, Prentice-Hall, London, 1989. 255
28. Thomas, L. D., "System Engineering the International Space Station," NASA Langley Research Center, *International Space Station Video Conference '97*, 1997. 251
29. Wiederhold, G., "Mediators in the Architecture of Future Information Systems," *IEEE Computer Magazine*, March, 1992. 256
30. Zave, P., Jackson, M.: "Four Dark Corners in Requirements Engineering," *ACM Transactions on Software Engineering and Methodology*, Vol 6, No. 1, January 1997. 248, 251, 255
31. Zave, P., Jackson, M.: "Conjunction as Composition," *ACM Transactions on Software Engineering and Methodology*, Vol 2, No. 4, October, 1993. 255, 257

Using the Model Paradigm for Real-Time Systems Development: ACCORD/UML

Sébastien Gérard, François Terrier, and Yann Tanguy

CEA / DRT-LIST / DTSI / SLA / L-LSP
F-91191 Gif sur Yvette Cedex France
(sebastien.gerard,francois.terrier,yann.tanguy)@cea.fr
Phone: +33 1 69 08 58 24, Fax: +33 1 69 08 20 82

Abstract. Industrial competition is intensifying and products are becoming ever more complex and software-intensive. Engineers must therefore face the challenge of being both business experts and advanced software developers. This situation urgently requires new ways of thinking about software development, with the primary objective of minimizing development efforts wherever possible, so that engineers can focus on the added value of their products. The latest OMG initiative, called MDA -- for "Model Driven Architecture" -- advances the idea of using a combination of the model paradigm and various underlying technologies, to offer solutions to this challenge. For real-time domain applications, engineers may already be using model-orientated approaches ([1], [2],). However, these are not yet entirely satisfactory, since they still require advanced real-time development skills. This paper describes the ACCORD/UML approach that was originally designed to lighten the daily workload of automotive engineers by relieving them of need for real-time expertise.

1 Introduction

Faced with intensifying industrial competition, engineers are increasingly being asked to focus on the market research and business development sides of their activities. This means finding ways to differentiate their products, making them more and more innovative than those of their competitors. It also entails further reduction in the famous time-to-market and constantly greater product complexity! In parallel, products are becoming ever more software-intensive, and thus also require high-level skills in software engineering. Today's engineers must therefore face the challenge of being both business experts and advanced software developers.

The automotive domain, for example, is very representative of such a situation. It is an area where competition is relentless and engineers must be constantly "creative". Moreover, new vehicle functions are relying more and more on software capabilities requiring advanced software engineering know-how. Since such know-how evolves quickly, it is also becoming more difficult for engineers to find time to update their

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 260-269, 2002.

software development skills . Functions such as ABS, navigation systems and engine control are illustrative of this state of affairs.

It is therefore urgent to define new techniques for software development that allow engineers to concentrate on their business approach. In response to such challenges, as stated in [3], one emerging solution could be to switch from code-oriented to model-oriented development of software intensive applications. In this respect, the model paradigm, where suitably supported by industrial toolkits, seems very promising as a means for helping engineers master the complexities of state-of-the-art applications.

OMG has performed a considerable amount of work around CORBA for defining standard middleware to manage the development problems associated with distributed systems in heterogeneous environments. Over the last five years, the OMG Unified Modeling Language has also become the "de facto" language for several methodologies ([4], [1], [5], [2], [6], etc.). Now that CORBA and UML have each developed well on its own, OMG is trying to assemble these two paradigms and has coined a new technology concept -- MDA ([7], [8] and [9]). But what actually lies behind this "revolutionary" concept?

First of all, MDA means "Model-Driven Architecture", and it is always teamed with two other important words in this context: "PIM" and its friend "PSM". The "P" in both these acronyms stands for platform and refers "to technological and engineering details that are irrelevant to the basic functionalities of a software component" [8]. A PIM, "Platform Independent Model", is also a work tool in which engineers only define and manipulate elements that are fully independent of any software implementation technology. The obvious advantage of this feature is that it should be easier to port/transform a business model into different operational environments relying on different implementation technologies. Indeed, once such a high level model, in this case the PIM, has been constructed, it becomes a PSM, i.e. a "Platform Specific Model". At this stage, implementation technologies are chosen and the PIM is "implemented" on or via a PSM. For example a PIM may be transformed either into a PSM1 using technologies such as C++ and VxWorks, or into another configuration, PSM2, based on Java and Windows NT.

The first significant result of the MDA paradigm for engineers is the possibility for them to build application models that can be conveniently ported to new, emerging technologies - implementation languages, middleware, etc.- with minimal effort and risk.

In the real-time application area, this model-oriented trend is also very active and promising. Currently, there are four main model-oriented industrial approaches supported by tools: UML-RT used with Rose-RT, ROPES with Rhapsody, ARTiSAN and UML/SDL with the Tau UML/SDL suite.

Within UML-RT, an application is seen as a set of entities called "capsules" which support logical concurrency. These capsules have a state machine as behavior specification and may exchange signals to communicate. Models built in this way are said to be executable, meaning that at any moment in design, it is possible to produce an executable application matching the UML model. In this case, the mapping is achieved via code generation.

For ROPES and ARTiSAN approaches, real-time application modeling is a 3-stage process: i) building a □functional□model with class and state diagrams; ii) building a specific tasking model with class diagrams containing only active objects (~execution tasks); iii) describing the mappings between the two models. The main drawback of this "family" of methods is that it requires advanced real-time development skills to build the tasking model and map it with the □functional□model. While there are some "shortcuts" available ([10]) to facilitate this activity, no transformation rules are provided as could be done within a fully MDA-based approach.

The approach proposed by Telelogic is based on the use of both UML and SDL languages. It consists of building UML models at the analysis stages using active objects as concurrency supports and SDL within design-time. Reference document [11] defines modeling rules for mapping a UML-oriented model into an SDL-oriented model. When SDL models are finished, the engineer may generate code to produce an executable application.

All these methodologies may be considered as MDA-based approaches for mainly two reasons. Firstly, they clearly promote the model paradigm to develop applications; and secondly, they provide code generation taking into account structural and behavior specifications for model mapping to implementation languages such as C, C++, JAVA,□

Nevertheless, they do not exploit all the potentialities of MDA. Their application models are often only PSM-like for "executable" reasons. For modeling purposes, the user is thus led very quickly to resort, for an executable model, to a programming language such as C++, . Although action semantics have been standardized by OMG [12], there are still only a few tools that have integrated this feature, which allows building of executable models independently of any programming language.

While these approaches are usually based on a several stage process, they do not provide the refinement mapping rules that could facilitate application development and, above all, be highly useful in promoting seamless development processes.

Finally, the existing UML-based methods for real-time applications still require considerable knowledge of real-time software technology (and the different programming models promoted by these tools) to develop real-time systems.

The subject of this paper is then to present an approach called ACCORD/UML[1] dedicated to real-time application development. The audience for such a methodology includes engineers who are not □software□experts. Its aim is to provide both a method and the underlying tools in abstract enough form to enable specification and prototyping of real-time systems by non real-time experts.

As depicted in Fig. 1, ACCORD/UML relies on a basic three stage software development cycle that enables analysis, prototyping and testing. Progression from one to another of these phases is achieved by a seamless and iterative process of refining UML models[2]. For that purpose, the ACCORD/UML profile [13] defines specific

---

1 Parts of this article were taken from research conducted by S□bastien G□rard as part of his PhD thesis, in collaboration with the French car maker PSA; said research is now being partially supported by an European project under the 5th FRDP (Framework Research and Development Program): AIT-WOODDES (http://wooddes.intranet.gr/).

2 In the rest of this paper, due to space limitation, we will focus on mappings defined within the analysis phase.

packages of model mappings. This profile also contains all UML extension definitions introduced within the ACCORD/UML context to facilitate access by non-experts to real-time systems development.

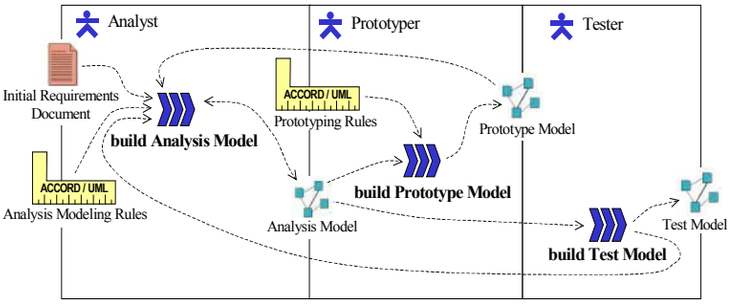


Fig. 1. Overview of the ACCORD/UML methodology[3]

The global model of an ACCORD/UML-based application is in turn broken down into three sub-models: an Analysis Model, a Prototype Model and a Test Model.

Within MDA, one important technological issue concerns mappings which may be derived in different configurations such as PIM-to-PIM, PIM-to-PSM or PSM-to-PSM [8]. The purpose of this paper is to focus on mapping and to demonstrate that it is feasible and efficient to employ a full MDA-based methodology that uses model mappings intensively throughout the development process.

The rest of the paper is then structured as follows: First a description of the Analysis Model building phase. Since this first model is made up of two submodels, emphasis is placed on how to switch from one model to the other in a PIM-to-PIM mapping situation. This aspect is then described in the second section The final section is devoted to conclusions on the efficiency of a full MDA-based approach and to potential future uses of the ACCORD/UML approach.

2 Analysis[4] Model Building Phase

This phase serves to describe what the system is expected to do. All model elements involved here are derived from the problem domain and from the physical constraints imposed on the system (e.g. Automotive, Telecom, etc.). The product of this modelling is then used first as input for the prototyping phase (see section entitled "Prototype Model building phase") to specify how the "what[5]" is to be performed. The Analysis Model is also used to build the test models that validate the application (see section entitled "Test Model building phase").

3 This figure uses the modeling artifacts defined in the OMG SPEM profile.

4 Use of the term "analysis" requires some clarification here. An "Analysis model" is a model that describes requirements and may also be called a "specification model". It should not be confused with "model analysis" that may refer, for example, to model validation.

5 i.e. results of analysis that specify what the system is expected to do.

The Analysis Model is split into both the following submodels:

- Preliminary Analysis Model (PAM) – The PAM is intended to specify the overall functions of the application, in very general terms, as well as its interactions with the environment. Throughout this activity, the application is only considered from an external viewpoint, in other words as a "blackbox" interconnected with its environment;
- Detailed Analysis Modeling (DAM) – once the PAM is built, the user zooms in on the blackbox, and also builds the DAM. To do so, s/he describes as thoroughly and accurately as possible the structural, interactional and behavioral aspects of the application.

2.1 The "PAM" Model

This first activity of the method thus consists of building a model called the Preliminary Analysis Model (in short PAM). This step plays a significant role in the project development cycle. It is the process activity during which product requirements (where they exist) can be reformatted as text and graphics that are easily accessible even to inexperienced users and also become formal[6] specifications. Moreover, if not already the case, model building also offers the system analyst opportunity to become familiar with vocabulary and concepts specific to the application domain s/he is working for. As depicted in Fig. 2, a PAM is made up of a dictionary, a use case model and a high-level scenario model.

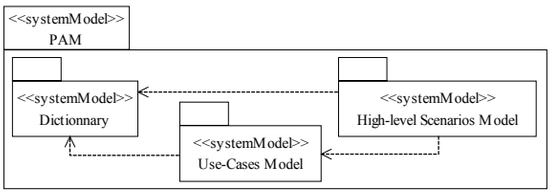


Fig. 2. A Preliminary Analysis Model

2.2 The "DAM" Model

The clear, unambiguous model of user needs that results from preliminary analysis still reflects primarily a functional view of the system. The aim of the subsequent phase is also to build an object-oriented model based on user requirements expressed essentially as use cases and sequence diagrams. The second objective within DAM building is also to move from "a black-box" to a "white-box" perspective, with the latter detailing the content of the system. Throughout this stage, however, the engineer must remember that s/he is modeling the "what" and not the "how" of the system. To put it clearly, s/he is modeling requirements for implementation, not the implementa-

6 Formal means here that the resulting model will be interpreted in the same way by any user. For our purposes, formal specifications are not mandatory written in a mathematical-based language.

tion itself! For this reason, it is proposed to organize global model construction around three dependant partial, but complementary and consistent "sub-models" (Fig. 3):

(i) Structural Model – it defines the general architecture (topology) of the application in terms of classes and the relations between them. Its modeling function is limited to the "class" level of the application, i.e. to specifying both local class properties and those affecting the application as a whole (which means accounting for all necessary and possible interactions between classes). The structural model is described in particular by UML class diagrams;

(ii) Behavioral Model – it defines the behavior of classes involved in the application. It is likewise concerned with the "class" level only, i.e. with specifying class behavior. The behavioral model introduces two object views: that of the protocol, which specifies the global behavior (also known as the "life cycle") of the object; and the "triggering" view, which accounts for reactive behavior of objects such as reactions to received signals, periodic behavior, etc. The model may also describe the behavior of class operations;

(iii) Detailed Scenarios Model – it is concerned with application "instances" and defines message passing between these various instances for the purpose of performing a given task. The interaction model is specifically described by UML use case and sequence diagrams. It has an additional facet to enable specification of application start-up and initialization features of an application. The application installation is likewise specified via the interaction model, using a specific sequence diagram dedicated to this aspect.

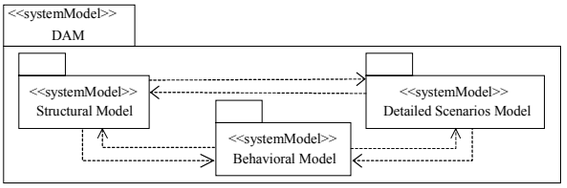


Fig. 3. A Detailed Analysis Model

3 From PAM to DAM, the Structural Viewpoint

The structural aspect of a DAM is based on a layered model especially designed to clearly identify the connections between a system and its environment. This model thus enables structuring the system in such a way as to facilitate component development by placing emphasis on specification of both "provided" and "required" interfaces. The following issues relating to component development are thus emphasized:

how a component is to be used, through clear definition of provided interface

what requirements are set for use of the component and how it is plugged into the environment needed to run it, through clear definition of a required interface?

To facilitate construction of an application in terms of reusable components, a generic software architecture is needed that emphasizes separation of the core of the system from its interface with the environment. This architecture is divided into the following five packages (Fig. 4).

The *ProvidedInterfaces* package describes active interaction points (i.e. where and how the environment "stimulates" the system). All the elements of this package play sensor roles in the system, i.e. they relay information from the environment to the system.

The lowermost package, called *RequiredInterfaces*, depicts the connection points at which the system interacts with its environment and models the environmental interface required for the system to operate. This package defines the specification that the environment needs to run the developed component. The elements in this package play an actuator role, i.e. they relay information from the system to its environment.

The middle layer describes the core of the system and is also called the Core package. It incorporates the actual inside of the application model. This package is a □zoom-in□on the single class introduced into the PAM model to represent the whole system.

For reasons of organization (to avoid confusing different concepts and preclude cyclic dependency among the packages), all of the signal definitions are grouped together in separate stereotyped □Signal□ packages[7] depending on their scope. As described in Fig. 4, an application may receive/send two kinds of signal: internal signals whose scope is the core package (these are considered as signals internal to the application); and external signals that may be shared with external systems/components located in the same namespace.

A last specific package can be added to this template of structure. It is shown on the left hand side of the figure. It is stereotyped □DomainTypes□and is introduced for the purpose of "collecting" particular business types. This is because, very often, the analyst is obliged to make design choices to model the requirement. For example, an automotive engineer may use a speed type. In this case, the first reflex of the analyst may be to say: □Let□s have an integer or a float□ □. In our approach, s/he will then define a new type in this last package (e.g. □Speed□type) and does not need to make premature design choices.

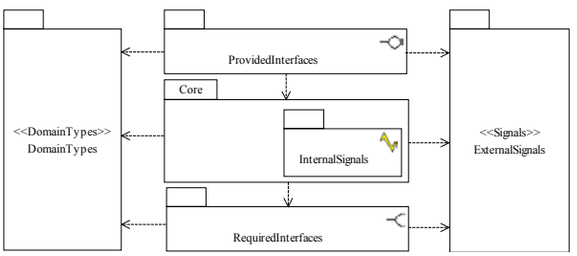


Fig. 4. Generic Structural Model of an ACCORD/UML-based Application

Use of two specific interface packages and the choice of orientations for links between them are intended to ensure clear structuring and identification of dependencies between the developed system and its environment. Such design is necessary to facilitate system integration into an existing context and enhance the reusability of the application analysis models.

7 A stereotyped □Signal□ package is also depicted via the icon .

Regarding specific templates for structural models, the ACCORD/UML method defines, within its profile [13], the following modeling rules:

- *A package stereotyped «Signals» must contain only signal-type elements.*
- *The package stereotyped «DomainTypes» must only contain model elements stereotyped «type».*
- *In the requirements analysis phase, stereotyped packages « ProvidedInterfaces » and « RequiredInterfaces » contain only interface classes.*

During this step, some specific classes of the application may be identified automatically from the PAM. These are the System/Environment interface classes, also called interface classes. By applying the following modeling rules, the analyst populates both interface packages of the application in a systematic way:

- *Each actor identified in the Use-Cases Model built during preliminary requirements analysis (i.e., the PAM) results in identification of a corresponding interface class in the model generated during detailed analysis (i.e., the DAM). All of the classes introduced in this way are interface classes*[8] *(and are therefore either given an "interface" stereotype or depicted as circles with class name labels) and are assigned the same names as the actors they refine. Furthermore, if an actor is stereotyped as « active »*[9]*, the corresponding class is positioned in the provided interface package. If, on the other hand, it is stereotyped as « passive », it is included in the required interface package. An actor element in the PAM is linked via a dependency link stereotyped «refine» with its matching interface class in the DAM. Moreover, the direction of the dependency is from the interface class toward the actor.*

The ACCORD/UML method is supported by a tool that allows implementation of the UML profile concept and, in particular, of modeling rules defined within a methodology. Mapping rules have been developed using the J language of Objecteering's UML Profile Builder [18]. This tool ensures extension of Objecteering capabilities to support a specific method, whether using standard UML extension mechanisms (stereotypes, tagged values, etc.) or adding behaviors using J language, e.g. by implementing model transformation rules as defined above.

4 Conclusion and Outlook for the Future

As stated in the introduction to this paper, one of the main issues for engineers in the coming years is a need for "double" expertise, in both business and software engi-

8 *UML* definition : "An interface is a named set of operations that characterize the behavior of an element."

9 Active and passive actors are determined thanks to the following modeling rule: "If an actor's role in a sequence diagram is only to output messages and receive responses to these messages, said actor is considered to be active and is therefore stereotyped as such. If, on the contrary, it serves only to receive messages sent by the system, it is said to be passive (even if it responds to the messages received) and is stereotyped accordingly."

neering, coupled with the increasing complexity and competitiveness in the business environment. The model paradigm appears as a possible solution for meeting this challenge. Throughout this paper, we have tried to demonstrate the main tenets of the AIT-WOODDES method with regard to MDA model progression. Two kinds of mapping are introduced: model refinement (e.g. PAM-to-DAM); and external model mapping (e.g. code and test generation). The main contribution of ACCORD/UML methodology is firstly its intensive use of model mappings at all stages in the development process, and secondly, the sets of rules defined within the underlying profile to facilitate model building/refinement at each of these stages.

The paper gave three examples to illustrate this point and demonstrate the efficiency of such a method. A PIM-to-PIM transformation was first depicted to demonstrate an internal model progression. This part of the paper described how to synthesize a draft of the DAM from the PAM.

For executable building, code generation rules are specified and implemented in an industrial UML-based tool. The generated code then allows the user to build a multitasking application without knowing anything about Real-Time Operating Systems.

Test model building relies on a formal tool, called AGATHA, that is able to compute all the symbolic execution paths of an application and generate an exhaustive set of behavioral test sequences. Model transformation is also intensively used. A file is first generated to the external tool, then the results are analyzed and retransformed into UML models, i.e., sequence diagrams.

The main point to be made here about model mapping rules is that up until now, these rules were defined informally (in natural language) and implemented with the proprietary language of an industrial UML-based tool that also depends on the tool support. The main requirement for incoming work will be to formalize these mapping rules within the ACCORD/UML profile so that they can be imported into any tool capable of accommodating them and that the methodology proposed here can be easily integrated into another UML-based tool that supports/imports UML profiles.

References

[0] K.-D. Vöringer, "Software and the Future of the Automotive Industry," in Proc. 2nd ITEA Symposium, Berlin, Germany, October 12th, 2001. http://www.itea-office.org/

[1] B. Selic and J. Rumbaugh, "Using UML for Modeling Complex Real-Time Systems," ObjecTime Limited 98.

[2] B. P. Douglass, "Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns," Addison Wesley, 99.

[3] J. Bézivin, "From Object Composition to Model Transformation with the MDA," in Proc. of TOOLS USA, Santa Barbara, August 2001.

[4] A. Lanusse, S. Gérard, and F. Terrier, "Real-Time Modeling with UML: The ACCORD Approach," in Proc. "UML98": Beyond the Notation, Mulhouse, France, 98.

[5] F. D'Souza and A. Wills, "Objects, Components, and Frameworks with UML: the CATALYSIS Approach," vol. 1, 1999.

[6] S. Gérard, P. Petterson, B. Josko, "Methodology for developing real time embedded systems," IST-1999-10069, 2002.
[7] R. Soley and t. O. S. S. Group, "Model Driven Architecture (Draft 3.2)," OMG, White paper November 27, 2000 2000.
[8] J. Miller and J. Mukerji, "Model Driven Architecture (MDA)," OMG, Specification July 9, 2001 2001.
[9] J. Siegel and t. O. S. S. Group, "Developing in OMG's Model-Driven Architecture," OMG, White paper Revision 2.6, November, 2001.
[10] M. Awad, J. Kuusela, and J. Ziegler, "Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion," Upper Saddle River, NJ 07458, USA: Prentice Hall, 96.
[11] ITU-T, "Recommendation Z.109: Languages for telecommunications applications - SDL combined with UML," ITU-T, Geneva Novenber 99 99.
[12] OMG, "UML 1.4 with Action Semantics," OMG ptc/02-01-09, 2002.
[13] S. Gérard, "The ACCORD/UML profile," CEA-LIST, Gif sur Yvette, Internal report 2002.
[14] D. Lugato, N. Rapin, and J.-P. Gallois, "Verification and tests generation for SDL industrial specifications with the AGATHA," in Proc. of Workshop on Real-Time Tools, CONCUR'01, 2001.
[15] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, "Object-Oriented Modelling and Design," Prentice Hall, 91.
[16] I. Jacobson, M. Christerson, P. Jonson, and G. Övergaard, "Object-Oriented Software Engineering: A Use Case Driven Approach," 1992.
[17] M. Broy, "Requirements Engineering for Embedded Systems," in proc. of FemSys'97, 1997.
[18] P. Desfray, "Profiles UML et langage J : Contrôlez totalement le développement d'applications avec UML," Softeam, Paris, white paper, 1999.

Generating Enterprise Applications from Models

Vinay Kulkarni, R Venkatesh, and Sreedhar Reddy

Tata Research Development and Design Centre
54, Industrial estate, Hadapsar, Pune, 411 013, INDIA
{vinayk,rvenky,sreedharr}@pune.tcs.co.in

Abstract. For developing large and complex applications, industrial practice uses a combination of non-formal notations and methods. Different notations are used to specify the properties of different aspects of an application and these specifications are transformed into their corresponding implementations through the steps of a development process. The development process relies heavily on manual verification to ensure the different pieces integrate into a consistent whole. This is an expensive and error prone process. We present a set of notations for specifying the different layers of a software architecture and a method for transforming a specification into an implementation. Models defined using these different notations are instances of a single meta model. This provides a means to unify the specifications of different layers and leads to a simple and elegant implementation method. The method has been used extensively to construct medium and large-scale enterprise applications.

1 Introduction

Faced with the problem of developing large and complex applications, industrial practice uses a combination of non-formal notations and methods. Different notations are used to specify the properties of different aspects of an application and these specifications are transformed into their corresponding implementations through the steps of a development process. The development process relies heavily on manual verification to ensure the different pieces integrate into a consistent whole. This is an expensive and error prone process. In this paper we address the problem of managing scale and integration of application development.

Scale and complexity are addressed by breaking down the problem along different axes – functional, architecture and development process (Fig. 1). Functional break up results in various components. For a layered architecture the application is split up so that each piece implements the solution corresponding to a layer in the architecture. Different phases of the development process determine the properties of the application that are to be specified (implemented) during a particular phase. For example, a banking system may be broken down into different functional components like *Foreign Exchange*, *Retail banking* etc. A functional component like *Retail banking* will have a *User Interface layer* describing the way a user interacts with the system, an

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 270-279, 2002.

Application layer implementing the business functionality and a *Database layer* making information persistent. A development process consisting of phases such as *Analysis*, *Design* and *Implementation* will implement different properties of a layer. The Analysis phase for the user interface layer of *Retail Banking* will define the screen layout. The Design phase will identify the controls to be used and define the user interface standards. The Implementation phase will code the user interface in the chosen platform.

Analysis	UI prototype	UML diagrams	
Design	GUI standards	Design Strategies	ER diagrams + Table design
Coding	JSP implementation	C++/Java code	RDBMS Implementation
	User Interface(UI)	Application	Database

Fig. 1. Break up of application based on development phases and architecture layers

It is a common practice to split up an application into different functional components. With regards to architectural layers and development phases there have been various unrelated attempts. Entity Relationship (ER) modeling [3] is popular for specifying information content of an application. Unified Modeling Language (UML) [2] provides different notations without clearly stating the relationship between specifications defined using these notations. A given UML notation can be used to express different properties. For example, an activity diagram can be used to specify control flow within a function as well as a business process. Therefore, a UML specification cannot be unambiguously transformed into an implementation. There is a need to define a set of notations having well defined semantics that will enable unambiguous transformation of a specification to its corresponding implementation. The set of notations should allow specification of all aspects of interest of an application it should be complete with respect to the aspects of interest. In this paper we present a set of notations and a method for constructing an enterprise class application using these notations for specifying the different architecture layers. Models defined using these different notations are instances of a single meta model. The meta model specifies the structure and constraints between the related model elements [4]. This provides a means of unifying the specifications of all the properties of the application that is constructed using these models leading to a simple and elegant implementation method. The method has been used extensively to construct medium and large-scale enterprise applications.

We first describe a possible set of architecture layers in which the development of an enterprise application can be decomposed. We propose models to specify these layers and the relationships between them. We show how each layer can be independ-

ently transformed into its implementation without compromising the integrity of the application. For clarity, we have used meta-modeling approach to illustrate the idea in a diagram. We conclude with advantages of the approach and some problems that remain to be solved.

2 The Use of Modeling for Application Development

The development of an application starts with an abstract specification A which is to be transformed into a concrete implementation C on a target architecture [5]. The target architecture is usually layered with each layer representing one view of the system.

The modeling approach constructs A using different abstract *views* $A_1 \ldots A_n$, each defining a set of properties corresponding to the layer it models. A view, A_i, is an instance of a more general structure that can be represented as a model M_i, e.g. user interaction model for sequences of interactions between the user and the system or an entity-relationship model for representing data. Note that A is usually not available separately: A is used here to represent the composition of the views $A_1 \ldots A_n$.

A view is the means by which one set of abstract properties are specified and implemented through a corresponding set of transformation mechanisms, i.e. by transforming each A_i into a corresponding implementation C_i. The application level composition of $C_1 \ldots C_n$ gives C, which is the intended implementation of A. Each A_i can be transformed into C_i manually, but this is then required to be done for each such application. Instead, we make use of the model M_i, of which A_i is an instance, and implement general transformations at the model level. These transformations can be applied to all instances of M_i. Defining transformations at the level of M_i, rather than A_i, makes it possible to scale-up the method to handle large programs. For example, a transformation can be specified from Class diagram to Java classes. This transformation can be applied to generate Java classes for any application.

3 Architecture of a Client-Server Application

An enterprise application is implemented across three architecture layers □user interface, application functionality and database. Each layer is implemented on a different platform supporting different primitives. For example, User interface platforms like Visual Basic provide windows and controls as implementation primitives. Application logic is implemented in a programming language like Java with classes and methods as the primitives while the database layer is implemented using tables and columns in a relational database system. These three layers are implemented independently and combined later to get an implementation C.

A good specification should be capable of refinement and reflect the structure of the implementation. It should use the same vocabulary across different development phases. For example, if the implementation is using Classes the Analysis should be expressed as class diagrams. This enables easy transition from analysis to code. Since the implementation primitives are different for each layer we also need a different specification notation corresponding to each layer. The specification should also clearly bring out the relationships between the different layers. Having separated the

notations we can now define corresponding models and transformation on each model. These transformations can be carried out independently for each layer. The relationships between models specify part of the invariance to be satisfied by individual transformations. The transformation of the relationships between the layers will guarantee that the different C_is will integrate to give a consistent implementation C.

The following sections describe the different models Application, User Interface and Database, of a client server application in greater detail. For brevity, we have left out many details from each of the models.

3.1 Application

The business proess and information content of the application are modeled in the analysis phase. A business process models application as a set of tasks and defines the order in which these tasks are executed. Each task is implemented by a method. The application layer implements the business process, the business logic and business rules. The business logic is best modeled using classes, attributes, methods and associations between classes. This layer can be specified as an instance of the model in Fig. 2. Business logic is typically coded in a programming language.

Example: A banking system allows a user to open and operate an account with a bank. Two classes corresponding to this system are User and Account. An association between User and Account specifies the account belonging to a user. Account number is an attribute of Account and name is an attribute of User. The account opening process involves filling up an account opening form, verification of the form and approval of the form. A user can operate the account only after it is approved.

3.2 User Interface

A user interacts with an application through its user interface. The user feeds in information using forms and browses over available information using queries and reports. Forms, queries and reports are implemented in a target platform using standard

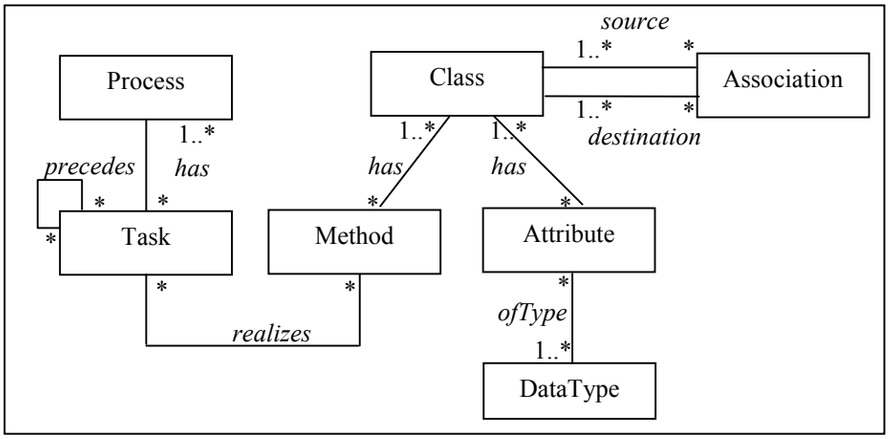


Fig. 2. Model for application layer

graphical user interface (GUI) primitives windows, controls and buttons. A window is a unit of interaction between the user and the system and is composed of controls and buttons. A control accepts or presents data in a specific format. The user can perform a specific task by clicking on a button.

Example: The user of a banking system needs a query window to inquire about past transactions and the current balance. She also needs a form window to withdraw money from her account. These windows will use appropriate controls to represent account number and date of transaction.

The user interface is best specified in terms of the windows in the application, data to be shown in each window, controls to be used to represent this data, possible navigation between windows and actions that can be performed. The core of a model for such a specification is as shown in Fig 3. In the figure a UIClass represents a logical grouping of the data to be shown in a window. Each UIClass represents a view of the application data. The association *mapsto* between UIAttribute and Attribute defines the view. This enables a transformation to represent value of the Attribute on the Window correctly. This also ensures the user can enter only values that are valid for the attribute of a class. The association *calls* between Button and Operation enables transformation to ensure type-correct invocation of operation. This also ensures the right set of objects get created and passed as parameters to the method invocation. The *mapsto* association enables a transformation to copy the right values from window to the parameter objects.

Additionally for a user interface to be good a particular data type should be represented by the same control in all the windows. In the banking system the same format should be used for all dates in all the windows. Similarly the same format should be used for account number in all the windows. An association between data type and control, as shown in Fig, 4, will allow specification of such GUI standards.

Additionally for a user interface to be good a particular data type should be represented by the same control in all the windows. In the banking system the same format should be used for all dates in all the windows. Similarly the same format should be used for account number in all the windows. An association between data type and control, as shown in Fig, 4, will allow specification of such GUI standards.

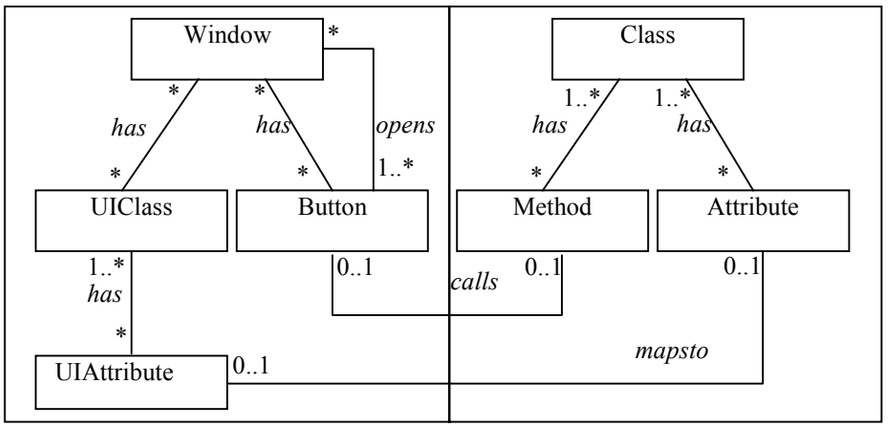


Fig. 3. Model for user interface layer

3.3 Database Layer

The database layer provides persistence for application objects using an RDBMS, access to these tables based on primary key and arbitrary predicates, and an object oriented view of these accesses to application layer.

In a relational database, the schema is made up of tables, consisting of rows and columns, where each column has a name and a simple data type. In an object model, the equivalent to a table is a class consisting of attributes and methods. A row in a table contains data for an instance of a class. Therefore, the mappings essential to object/relational integration are between a table and a class, between columns and attributes, and between a row and an object. This is shown in Fig 5.

Example: The persistent information for the banking system will include details about accounts and users. Two tables User and Account implement this persistent information. These tables have columns corresponding to user name and account number. The association between a user and an account is implemented by having account number as a foreign key in the User table and a primary key in the Account table.

Similar to the *mapsto* association of User Interface model, the *mapsto* association between Attribute and Column ensures type correctness. The *implements* association allows correct coding of class associations using appropriate Primary and Foreign keys. This association uniquely identifies the related classes and the tables. These classes and tables must be related. Such constraints can be specified in the meta model.

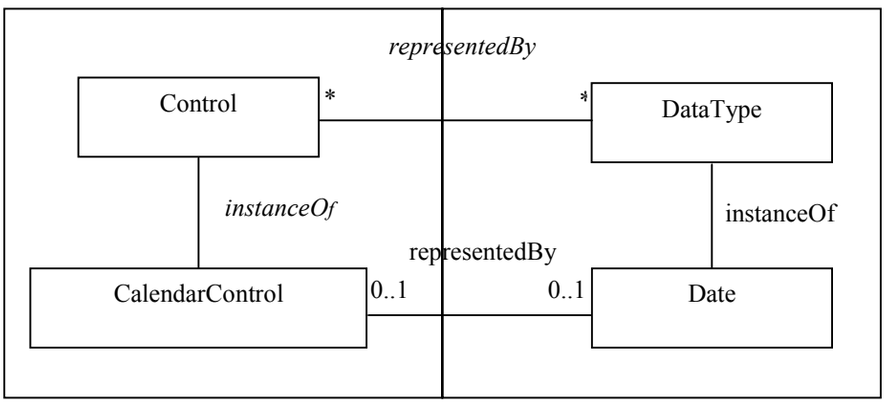


Fig. 4. A model for specifying GUI standards

4 Model Validation

Modeling of the application helps identify errors early in the development cycle. Associated with every model are a set of rules and constraints that define validity of its instance. These rules and constraints could include rules for type checking and for

consistency between specifications of different layers as well as across different phases. Some of the validation rules for models presented so far are:

- User interface should allow specification of all Tasks in the business process and be consistent with the *precedes* relationship between the Tasks
- User interface should display data that is consistent with respect to the parameters being passed to the operations invoked from the Window
- Database layer should ensure that *implements* association is implemented in a consistent manner. For example, the 1:M association between classes User and Account should be implemented by making the primary key of User table as foreign key of Account table.

5 Integration

We have illustrated the advantages of having different notations to specify the different layers of an enterprise application. Corresponding to these specifications are the three models - Application layer model (AM), User interaction model (UIM) and Database layer model (DM). These models represent different views of the system. Specifying these models and the relationships between them as an instance of a single meta model (MM) ensures consistency of the application. The meta model specifies structure as well as constraints to be satisfied by the instances of related model elements. Having a single meta model ensures that instances A_1, A_2 and A_3 of models UIM, AM and DM respectively can be transformed into corresponding implementations independently. These transformations can be performed either manually or using code generators. If individual transformation implements the corresponding specification and its relationships with other specifications correctly then the resulting im-

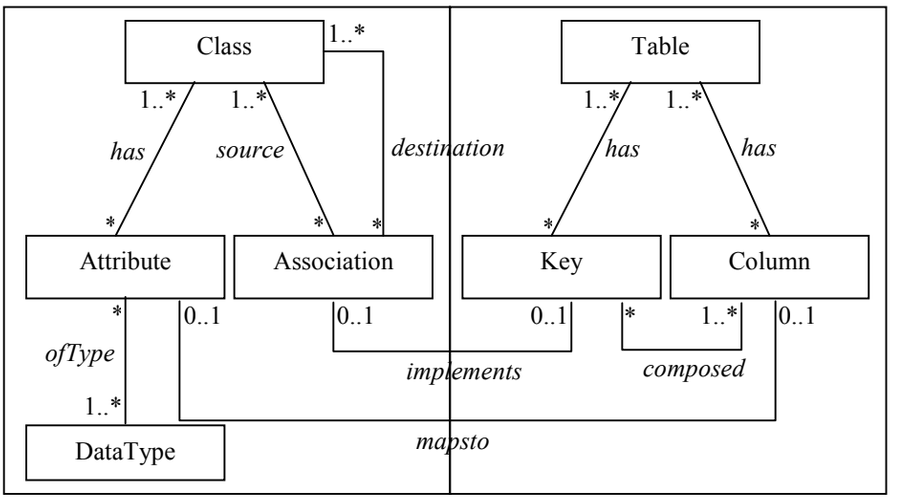


Fig. 5. Model for database layer

plementations will glue together giving a consistent implementation C of the specification A. Fig. 6 depicts this framework.

For the banking system, the specifications are as shown in Fig. 7. The association *has* between class User and class Account is implemented in the database layer by the column AccNo that is the Primary key in Account table and Foreign key in User table. Click of Deposit button invokes Deposit method of the corresponding Account.

The above example illustrates the advantages of using different notations to specify the different layers of an application. Making these specifications into instances of a single meta model allows us to specify the relationships between the different specifications. The notations proposed have well defined semantics. These properties allow specifications to be independently transformed into implementations that are guaranteed to integrate into a consistent whole. The ability to develop an application by specifying each layer separately and transforming each specification into corresponding code addresses the problem of scale. In the following section we describe a tool that performs these transformations automatically.

6 Model Driven Software Development Environment

The approach described above is well suited for model driven software development. This approach is realized in MasterCraft - a model driven software development environment [1] developed at Tata Research Development and Design Centre. This environment allows a user to model the three layers and the relationships between them independently (Fig. 8). This is achieved by extending the UML meta model with notations for User interface specification, Database specification and the relationships between all the three specifications. MasterCraft has code generators for each of the three layers *guimod*, *bxl* and *dmgen* that translate the specifications and relationships between them into an implementation. These code generators guarantee consistency across the different layers.

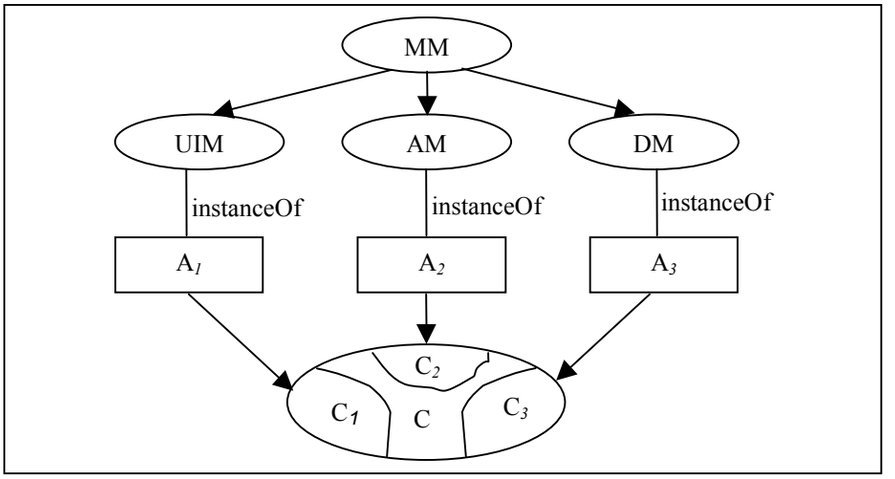


Fig. 6. Model based development and integration

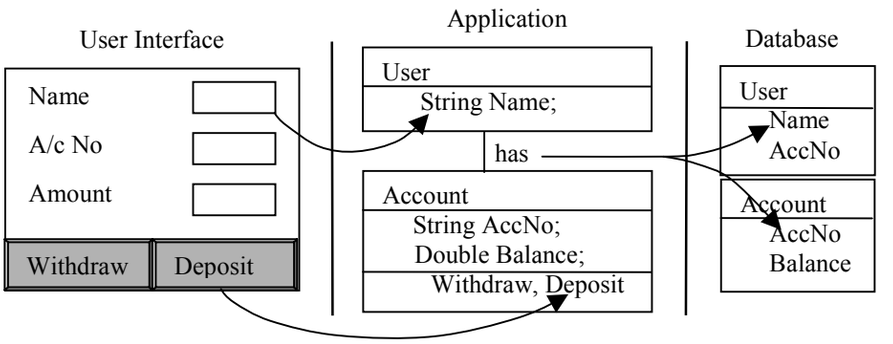


Fig. 7. Specifications for a banking system

The above approach has been extended to enable specification of other aspects of interest such as security, business rules and business process. The development environment also provides an automated development process. Several large applications have been developed using MasterCraft on different platforms.

7 Conclusions

We have presented the advantages of using different notations for different layers of application architecture. We have illustrated how having a single meta-model to describe the models corresponding to these notations and their relationships lends itself to an elegant implementation method. The implementation method allows independent transformations of specifications of the different layers and guarantees their integration into a consistent whole. The ability to develop an application by specifying and transforming each layer separately addresses the problem of scale.

Though we have illustrated the approach using a three-layer architecture, it lends itself to any architectural decomposition that has well-defined layers with well-defined relationships between them. The approach can be extended to support successive levels of refinement, with guarantees of integrity at each level of refinement until a level is reached that can be automatically transformed into an implementation.

The proposed approach can be used to realize an automated development process that integrates three orthogonal models Application model, Role model and Process model. This work will be presented in a separate paper currently under preparation.

The proposed approach can be extended to support notions like analysis and design patterns as first class model entities.

References

1. MasterCraft Component-based Development Environment Technical Documents, Tata Research Development and Design Center.
2. Object Management Group. Unified Modeling Language specification v 1.3. Available from www.rational.org/uml.

3. P. P. S. Chen. Entity-Relationship Approach to Systems Analysis and design, North Holland, 1980.
4. Sreedhar Reddy, Janak Mulani, Arun Bahulkar. Adex - A meta modeling framework for repository-centric systems building in COMAD 2000.
5. Sreenivas A, Venkatesh R and Joseph M,. Meta-modelling for Formal Software Development in Proceedings of Computing: the Australian Theory Symposium (CATS 2001), Gold Coast, Australia, 2001. pp. 1-11.

Tool Support for Aspect-Oriented Design

François Mekerke[1], Geri Georg[2], and Robert France[3]

[1] École Nationale Supérieure des Études et Techniques d'Armement, Brest, France
mekerkfr@ensieta.fr
[2] Agilent Technologies, Fort Collins, USA
geri_georg@agilent.com
[3] Colorado State University, Fort Collins, USA
{france,rta}@cs.colostate.edu

Abstract. In this paper, we describe the tool we plan to build in order to show the feasibility of aspect-oriented design, and demonstrate the advantages that it implies. This technique allows one to independently specify cross-cutting concerns and functionnal features of a system. The tool would then allow the weaving of the aspects on the model one after the other, creating a design model that would comply to all specifications, with additionally a high-quality architecture.

Introduction

Aspect-oriented design (AOD) localizes cross-cutting concerns to better manage design evolution and to enable reuse of these aspects (e.g. security or fault tolerance aspects) in multiple systems. A comprehensive system model can be obtained by weaving the model of essential functionality with aspect models. Design aspects allow one to understand and communicate cross-cutting concerns in their essential forms, rather than in terms of a specific system's behavior. Design aspect are also potentially reusable across different systems since they are not tied to any particular system. Policies and procedures intended to be applied across multiple systems can be expressed using aspect designs. Changes to any particular concern are made in one place (the aspect design), and effected by weaving the changed aspect into the models of essential functionality. This eases the management of design changes.

A number of authors have tackled the problem of defining and weaving aspects at the design level (e.g. [1], [2], [6]), but many of these approaches essentially result in wrapping additional functionality around an existing model by taking advantage of regular expression matching between aspect model elements and primary model elements. Therefore, the proper model factoring must already exist in order to apply the aspect in these cases. By contrast, our method of defining aspects using role models and weaving them into essential functionality through template as well as extension mechanisms is more flexible and therefore fewer constraints need to be placed on the models they are woven into.

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 280–289, 2002.

1 Background

1.1 Role Models

We use the concept of Role Models (see [5], [4]) to define aspects in an application-independent manner. Aspect properties are defined in terms of roles that can be played by model elements representing application-specific concepts. A model element conforms to a role if it possesses the properties defined in the role. Weaving an aspect defined by Role Models is essentially a model transformation process in which a non-conforming model is transformed to a conforming model (i.e., a model that incorporates the aspect). One can view a Role Model as a characterization of model element structures that incorporate the aspect. The UML is used as the design modeling notation in our work, thus the Role Models we developed are property-oriented characterizations of conforming UML models [5], [4]. A UML model element (e.g., a class or an association) that has the properties specified in a role can play the role, that is, it conforms to (or realizes) the role. A UML model is said to conform to (or realize) a Role Model (i.e., is a realization) if it consists of model elements that conform to the roles in the Role Model.

A design aspect (e.g., a security concern) can be modeled from a variety of perspectives. In our initial automated tool work, we focus on one aspect view (static) although we generally define an aspect using two views (static and interaction views). An aspect's static view (a Static Role Model, or SRM), focuses on the structure of the aspect. The interaction view (the Interaction Role Model, or IRM), focuses on the interactions that take place within the aspect. To facilitate weaving, Role Models are constructed in a manner that allows one to generate conforming structures from them. Weaving a Role Model into a UML model M, can involve (1) merging roles with model elements M, that is, modifying model elements in M so that they conform to the roles and (2) generating new model elements from roles and inserting them into M.

1.2 An Overview of SRMs

An SRM consists of classifier and relationship roles. Each role has a base that restricts the type of UML construct that can play the role. In general, association roles have multiplicity constraints expressed as templates of the form *[[n]]*, where n is constrained to be a range or a specific value. Multiplicities in a realization of a SRM containing these template forms can be obtained by substituting values for n that satisfy the constraints.

Each role defines properties that conforming constructs must possess. Two types of properties can be specified in a SRM role: Metamodel-level constraints are well-formedness rules that constrain the form of UML constructs that can realize the role, and Feature roles characterize properties that must be expressed in the conforming model elements. Metamodel-level constraints are constraints over the UML metamodel expressed in the Object Constraint Language (OCL) [7], [8]). For example, a metamodel-level constraint in a class role can

constrain conforming classes to be concrete with a class multiplicity that cannot exceed a value m. For examples of their use see [4].

Feature roles are associated only with classifier roles. There are two types of feature roles: Structural roles specify state-related properties that are realized by attributes or value-returning operations in a SRM role realization, and behavioral roles specify behaviors that can be realized by a single operation or method, or by a composition of operations. A feature role consists of a name, and an optional property specification expressed as a constraint template (omitted in the examples given in this paper).

Realizations of a feature role have properties that imply the property generated by appropriately instantiating the feature role's constraint template. For example, each behavioral role is associated with pre- and post- constraint templates that, when instantiated, produce pre- and post-conditions that must be implied by the pre- and post-conditions of realizing operations. (See [3] for examples of aspect models with behavioral roles.) In summary, a model element conforms to a role if: (1) it satisfies the metamodel-level constraint and (2) the constraints associated with its features imply the constraints obtained by appropriately instantiating the role's constraint templates.

1.3 Weaving Aspects into a Design Model

Weaving an aspect into a model involves identifying existing elements in the models of essential functionality that can play aspect roles. Weaving rules can add new elements, extend, or change existing elements so that all required roles in the aspects are present in the woven model. Weaving an aspect into a model involves:

1. Identify existing model elements that can play the aspect roles or can be modified to play aspect roles. This activity is carried outby the developer.
2. If no existing model element can play an aspect role, then a new model element is generated from the Role Model and added to the model of essential functionality. This is accomplishedin the following manner:
   - Create an instance of the role base that satisfies the metamodel-level constraints.
   - For each structural role, generate an attribute and associated constraints by substituting a name for the role name, and substituting conforming values for the template parameters of the constraint templates.
   - For each behavioral role, generate an operation and associated constraints by substituting a name for the role name, and by substituting conforming values for the template parameters for the pre- and post-condition constraint templates.
3. Modification of existing model elements may be necessary if they are to play the roles. Modification can involve adding new features (attributes or operations) to classes, changing association multiplicities, or moving features out of one construct and placing it in another. The required changes can be identified by matching the model elements with an appropriately instantiated form of the SRM.

2 Tool Design

2.1 Objectives

Our final objective is to provide tool support for aspect-oriented design, in order to allow developers to model their system with more ease. The advantages of using aspect-oriented design are the following:

- Simplified and systematic specification phase, through support for the separation of concerns principle: Architects can specify each cross-cutting concern independently of essential functionality.
- Potential automation of the integration of multiple concerns (weaving): A comprehensive architecture can be obtained by automatically weaving the aspects, one after the other, into the core functionality.
- High-quality architecture, through use of aspect patterns based on high-quality experience: Aspects can be used to create recognisable architecture, which are easier to understand.

2.2 Description of the Tool

The architecture of the tool is illustrated in Fig. 1. In the diagram, ovals represent data, rectangular boxes represent subsystems, and grayed boxes/bubbles represent elements that are out of the scope of our present effort.

The user must define the following inputs:

- An *Original Model* into which aspects will be woven (XMI format, created by an *Model Editor*);
- An *Aspect* (created by an *Aspect Editor*[1]), stored in an *Aspect Library*;
- *Weaving Strategies* (created by a *Weaving Strategy Editor*), which describe what aspects to weave into the model in particular conditions;
- *Mapping Instructions*, which define a correspondance between the roles of the aspect and the elements of the *Original Model* that are intended to play the roles.

These inputs are processed in two subsystems:

- An *Aspect Generator* creates a file defining the *Mapping Rules* that determine how the weaver will incorporate an aspect into a primary model;
- A *Weaver*, which modifies the primary model, using the aspect.

The result is a *Woven Model* whose properties are tested in a *Model Evaluator*.

[1] The *Aspect Editor* can be the same tool as the *Model Editor* as long as XMI descriptions of aspects can be defined

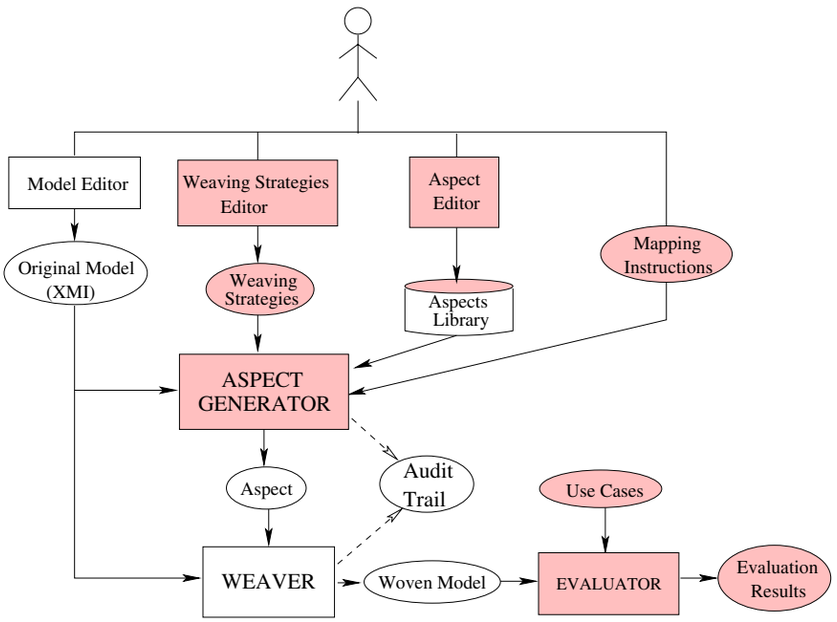


Fig. 1. General Architecture

2.3 Aspect Generator

The *Aspect Generator* is certainly the part of the tool that is the most difficult to conceive, since it has to collect data concerning the aspect and process them to produce a file containing mapping rules.

The elements needed as inputs for the aspect generator are:

- Analyzable forms of Role Models;
- Mapping instructions that describes which classes of the model play the roles defined in the aspect;
- *Weaving Strategies* used to select and compose aspects.

2.4 Weaver

The *Weaver* takes the *Original Model* and the *Mapping Rules* file generated by the *Aspect Generator* as inputs. It then processes those two files in order to provide the *Woven Model* and an *Audit Trail*. During this operation, the *Weaver* must preserve all the features of the original model, and add the features required by the aspect, which include new attributes and operations, but also new objects, thus new architecture.

In order to fulfill its missions, the *Weaver* must support the following operations:

– Create/delete an object or an attribute in a given context;
– Migrate attributes or operations from one class to another;
– Link a new object to the existing architecture.

For this, it will need to be able to compare the structures of both the *Aspect* file and the *Original Model*, to evaluate if the features of the model elements conform to the aspect, to add missing features and modify existing elements that do not conform to the aspect roles.

In order to have traces of what has been implemented (and then have the possibility to undo some operations), the *Audit Trail* records every operation performed by the *Weaver* and the *Aspect Generator*.

2.5 Evaluator

Once the *Woven Model* has been created by the *Weaver*, the *Evaluator* tests it, in order to :

– check its internal consistency;
– detect potential emergent properties;
– identify aspects conflicts.

The evaluation is performed by applying scenarios (described in *Use Cases*) to the *Woven Model*. These scenarios are system-specific, and should be chosen in such a way that all features of the system are tested in realistic situations. The *Evaluation Results* may be analysed to determine whether or not the system really meets the expectations of its architects.

3 Example

In this section, we demonstrate several inputs and results of the tool through a simple example. We discuss an *Original Static Model,* an *Aspect, Weaving Strategies, Mapping Instructions, Mapping Rules,* and the *Woven Model* produced by the tool. All models are shown as graphic diagrams for increased understandability. However, the tool takes model inputs in an XMI format, and creates XMI model output.

3.1 Original Static Model

A simple static class diagram that can serve as an *Original Model* input to the tool is shown on Fig. 2. The diagram is composed of four classes: *Manager, SystemMgmt, userList* and *userInfo. Manager* invokes the *addUser* method of *SystemMgmt* to add a user (u) to the system. User information is contained in one *userList,* while multiple *userInfo* objects contain information about specific users. Multiple *Managers* are associated with a single *SystemMgmt* object.

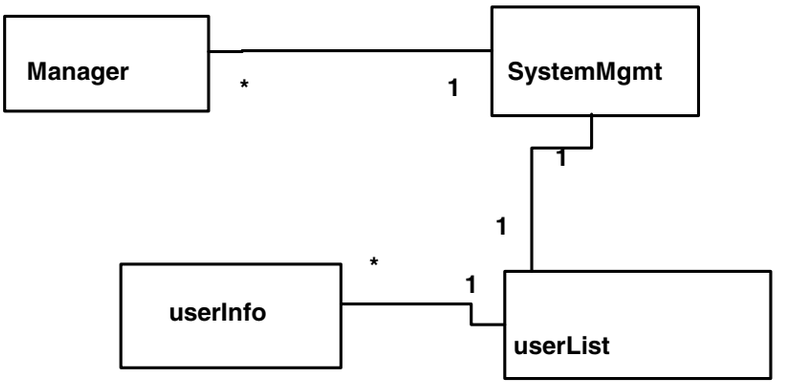


Fig. 2. Original Static Model

3.2 Aspect

Fig. 3 shows the structural view of an auditing aspect, called the Static Role Model (SRM). There are three classes in the auditing aspect SRM: *Invoker* invokes a method of *Invokee,* and *Log* records the outcome of the method.

This diagram also shows how portions of aspect models are parameterized, such as the multiplicities on the association between *Invoker* and *Invokee.* The values selected for these parameters during the weaving process must satisfy the OCL constraints shown in the diagram. The constraints state that the lower

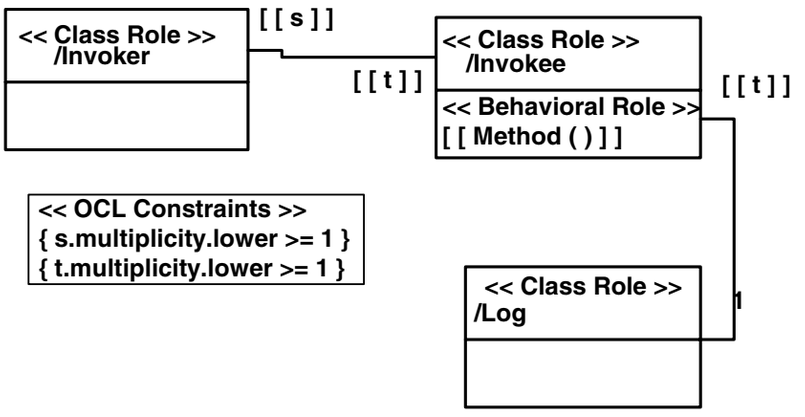


Fig. 3. Aspect

bound on the multiplicities must be at least one. The method that will be invoked is also parameterized (Method) and must be selected as part of the weaving process.

As indicated by the *Class Role* stereotype, *Invoker, Invokee,* and *Log* roles can be played only by UML Class instances.

3.3 Weaving Strategies

The weaving strategies for this example consist of a single directive, namely to weave the auditing aspect into the essential functionality model.

Weaving strategies become more important under two conditions: (1) if dependencies exist between multiple aspects that are being woven into a model, and (2) if different aspects need to be woven into a model under particular conditions.

For example, consider the case where auditing is woven into the essential functionality model shown in Fig. 2. This mechanism can be needed for a variety of reasons, among them the ability to recover from user errors. However, if the information flowing from *Manager* to *SystemMgmt is* sensitive, an authentication mechanism may need to be added to the system. In this case an authentication aspect also needs to be woven into the essential functionality model.

Weaving strategies can be used to direct the addition of an authentication aspect based on the presence of sensitive data, and they can also be used to

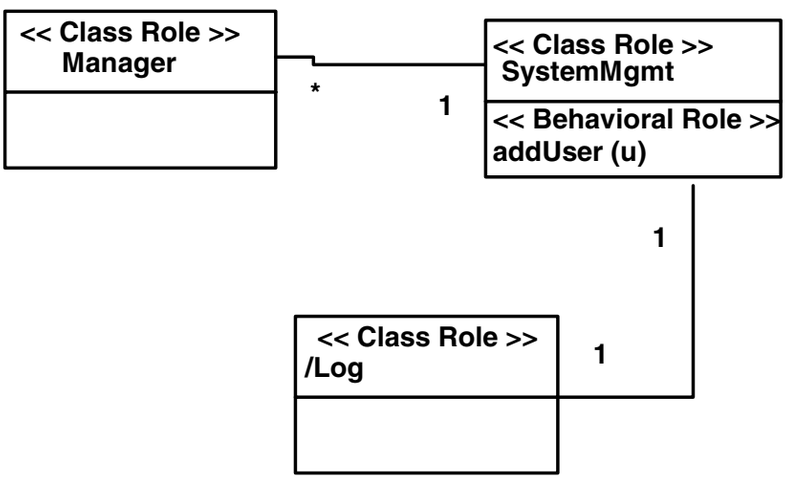


Fig. 4. Mapping Instructions

direct weaving order. In this case, authentication needs to be woven first, so that the additional authentication methods are audited.

3.4 Mapping Instructions

Fig. 4 shows the mapping instructions that define the correspondence between the aspect roles and the elements of the original model. This diagram shows that the *Manager* class in the original model will play the role of *Invoker.* Similarly, *SystemMgmt will* play the role of *Invokee,* and *addUser (u) will* play the role of Method ( ). There is no element in the original model that will play the role of *Log.* The multiplicities s and *t*of the association between *Invoker* and *Invokee* are selected as * and 1, respectively. These multiplicities satisfy the OCL constraints in the auditing aspect model.

3.5 Mapping Rules

Mapping rules are used to direct the weaving process. These rules state when model elements need to be added, deleted, or modified. The actual rules applied to each element in the code in order to create the final woven model are included in the audit trail. An example rule is that if an aspect role is not mapped to an original model element (e.g. *Log),* then it needs to be created in the woven model (e.g. *Logfile* in Fig. 5).

3.6 Woven Model

Fig. 5 shows the woven model. The aspect roles played by model elements are shown as stereotypes (e.g. <<*Invoker*>>). Original model elements are changed as needed to play these roles, and new model elements are added when no element in the original model can play a role (e.g. Log). (Note that behavioral roles (i.e. *addUser)* are omitted from this diagram for simplicity.)

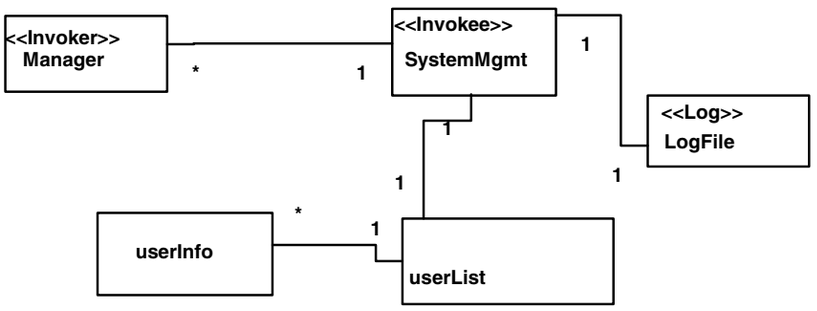


Fig. 5. Woven Model

Conclusion

As some of the concepts that we intend to integrate in the tool are not yet clearly defined, we have chosen to develop it incrementally and iteratively. We have thus begun with the building of a *Weaver*, which compares a hand-made *Mapping Rules* file and a *Model*, and modifies the structure of the *Model* so that it complies to the *Mapping Rules* specifications. The *Aspect Generator* and the *Evaluator*, which are the next steps of the development of the tool, are currently in the study phase, and should enter soon in the building phase.

References

1. S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Separating concerns throughout the development lifecycle. In *Proceedings of the third ECOOP Aspect-Oriented Programming Workshop, 1999.* 280
2. S. Clarke and J. Murphy. Developing a tool to support the application of aspectoriented programming principles to the design phase. In *Proceedings of the International Conference on Software Engineering (ICSE), 1998.* 280
3. R. France and G. Georg. Modeling fault tolerant concerns using aspects. *submitted to ISSRE02,* 2002. 282
4. R. B. France, D. K. Kim, and E. Song. Patterns as precise characterizations of designs. Technical Report 02-101, Computer Science Department, Colorado State University, 2002. 281, 282
5. R. B. France, D. K. Kim, E. Song, and S. Ghosh. Using roles to characterize model families. In *Proceedings of the Tenth OOPSLA Workshop on Behavioral Semantics: Back to the Basics,* 2001. 281
6. J. Suzuki and Y. Yamamoto. Extending UML with aspects: Aspect support in the design phase. In *Proceedings of the third ECOOP Aspect-Oriented Programming Workshop, 1999.* 280
7. The Object Management Group (OMG). Unified Modeling Language. Version 1.4, OMG, http://www.omg.org, September 2001. 281
8. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1999. 281

Model-Driven Architecture

Stephen J. Mellor[1], Kendall Scott[2], Axel Uhl[3], and Dirk Weise[3]

[1] Project Technology Inc., Tucson, AZ 85704, USA
[2] http://usecasedriven.com
[3] Interactive Objects Software
Freiburg, Germany

Abstract. This paper offers an overview of the basic MDA terms and concepts and the relationships among them, with the latter expressed in terms of the UML, and also an outline of a proposed software development process that would leverage MDA. The article also discusses the idea of "accelerated" MDA, which involves using one metamodel to do mappings of metamodels to multiple platforms.

1 Goals

Model-Driven Architecture (MDA) provides a framework for software development that uses models to describe the system to be built. The system descriptions that these models provide can be expressed at various levels of abstraction, with each level emphasizing certain aspects or viewpoints of the system.

The driving force behind the MDA is the fact that a software system will eventually be deployed to one or more platforms, used separately or together. Platforms are subject to change over time – and they change at different, typically higher, rates than the higher-level models of the system, which in turn tend to grow increasingly independent of the target platforms. This paper provides an introduction to the MDA's response to this conundrum.

2 Abstraction

The level of abstraction at which a certain amount of system detail is expressed typically determines the ratio of effort to the amount of detail that gets added. Best practices and accepted defaults allow for the definition of efficient and effective mappings to less abstract (more detailed) levels.

Relying on best practices continues to enable modelers to reach higher levels of abstraction. Tools are available that let one create a UML association between two business components, and subsequently map that simple line to literally thousands of lines of corresponding Java or C++ code and other artifacts such as configuration files and deployment descriptors. As we come to accept available best practices and proven

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 290-297, 2002.

defaults for model transformations, it becomes much easier to express certain elements of a system in a model, because we can do so at a higher level of abstraction and let a tool do the transformation into a more detailed specification.

3 Models and Metamodels

The OMG's Meta Object Facility (MOF) defines a model as an instance of a **metamodel**. A metamodel may make it possible to describe properties of a particular platform. In this case, the models that are instances of such a metamodel are said to be *platform-specific*, while models that describe a system at a level of abstraction, one that's sufficient to allow use of their entire contents for implementing the system on different platforms, are referred to as *platform-independent*. Figure 1 illustrates these relationships.

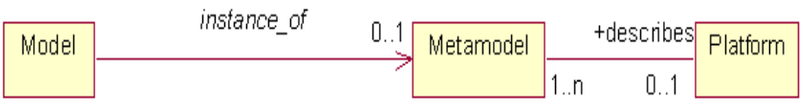


Fig. 1. Model, Metamodel, and Platform

4 Mapping between Models

Models may have semantic relationships with other models; for example, a set of models may describe a particular system at different levels of abstraction. It's desirable to have mappings between different but related models performed automatically. This makes it possible to express each aspect of a system at an appropriate level of abstraction while keeping the various models in synch.

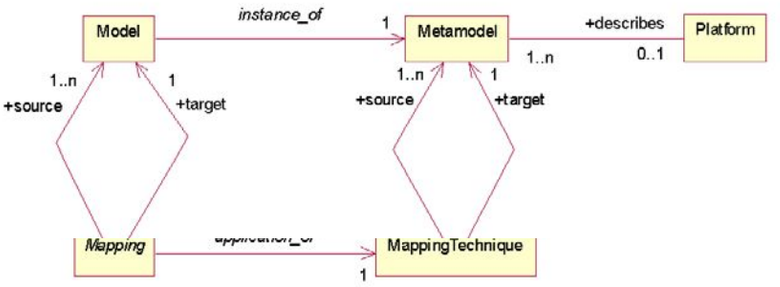


Fig. 2. Overview of MDA Modeling

A **mapping** between models is assumed to take one or more models as its input and produce one output model. The rules for the transformation performed by the mapping are described in a **mapping technique**. These rules are described at the metamodel level in such a way that they're applicable to all sets of source models. A modeler can automate a mapping technique by providing an executable implementation of its specification. Such an implementation allows for the automation of important steps of an MDA-driven process. However, this implementation isn't necessarily required as long as one can verify any manually conducted mapping against the specification that the mapping technique provides. For example, a mapping technique that describes how to map a UML model of a Java application to a corresponding Java source code model would have rules such as "A UML classifier maps to a Java class declaration, where the name of the class matches the name of the classifier." Figure 2 illustrates these concepts.

5 Platform Stack

A **platform** is the specification of an execution environment for models. A platform is relevant in the context of MDA only if there is at least one realization of it – in other words, an implementation of the specification that the platform represents. Note that a realization can in turn build upon one or more other platforms. In theory, this **platform stack** can extend down to the level of quantum mechanics, but for our purposes, platforms are only of interest as long as we want to create, edit, or view models that can be executed on them. Figure 3 illustrates these concepts in the context of models and metamodels.

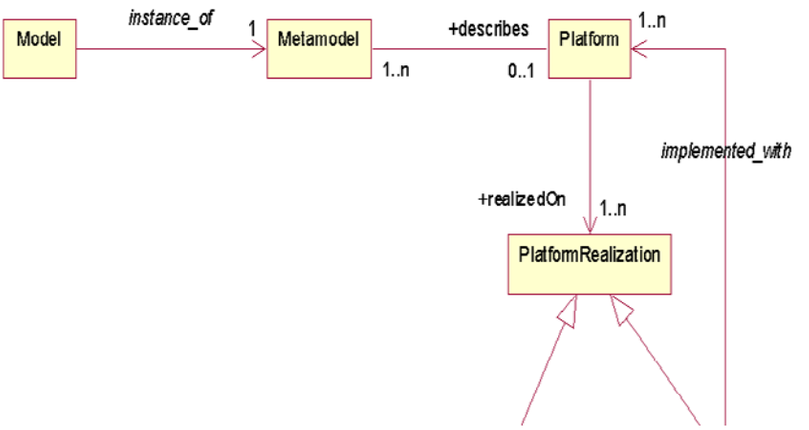


Fig. 3. Models, Metamodels, and Platforms

6 Annotating Models

MDA must support incremental and iterative development. This means that mappings between models must be repeatable. So, if a mapping requires input in addition to the source models, this information must be persistent. However, it mustn't be *integrated* into the source model, because it's specific to the *mapping*, and several different mapping techniques may exist, each of which require different additional inputs. Integrating the additional mapping input with the model would make the model specific to the corresponding mapping technique, which is not desirable.

These additional mapping inputs take the form of **annotations**. A mapping may use several annotations on the source models; conversely, an annotation may cater to several different mappings.

Just like a model is an instance of a metamodel, an annotation is an instance of an **annotation model**. This model describes the structure and semantics of the annotation. A mapping technique, therefore, specifies the annotation models of which it requires instances (annotations) on the instances of its source metamodels. If a mapping technique can use more than one annotation model for a single source metamodel, then one can reuse annotation models for several different mapping techniques. This, in turn, renders the corresponding annotations reusable for the different corresponding mappings. Figure 4 illustrates these concepts.

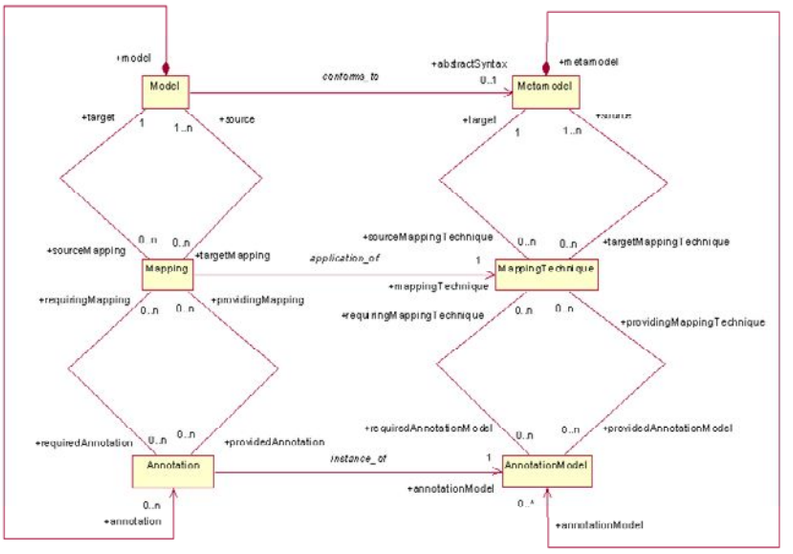


Fig. 4. Model Refinement Process

Figure 5 illustrates the idea that annotating a model for different mappings leads to different **platform-specific models** (**PSMs**). The source model for these mappings is a **platform-independent model** (**PIM**) with regard to the target platforms *A* and *B*.

The annotations don't pollute the PIM, which allows the PIM to be mapped repeatedly to two different PSMs.

7 Representing Models

A model has to be represented in some way. For example, the model of a Java source is typically represented in an ASCII file, while a UML model could be represented in the graphical UML notation or in ASCII using UML textual notation. While these representations are suited for editing by humans, there may be other representations of the same model that are better suited for transformation through programs – think of a Java compiler transforming the ASCII Java source into an abstract syntax tree before continuing with the semantic analysis, or a generator designed to transform a UML model into something else would use a representation of the model that provides a MOF-compliant interface such as JMI.A representation of a model is a model in itself, and therefore is an instance of a metamodel. For example, the ASCII representation of a Java source complies with the ASCII metamodel (which simply provides an alphabet). Figure 6 shows how the representation relationships among models can be described in terms of models, metamodels, and mapping techniques.

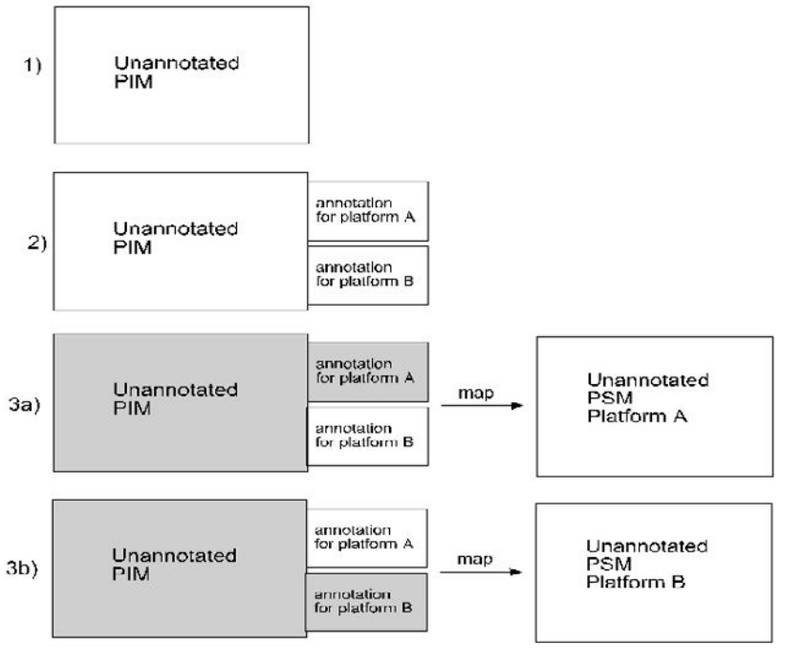


Fig. 4. From PIMs to PSMs

8 Outline of MDA-Compatible Software Development Process

The following seven process steps, taken together, offer a simple yet robust way to incorporate MDA into a software development project.

1. Identify the set of target platforms
2. Identify the metamodels that you want to use for describing models for these platforms, and also the modeling language/profile in which you'll express your models.
3. Find proper abstracting metamodels from the ones that are aligned along the lines of expected platform changes (with the goal being platform independence).
4. Define the mapping techniques you'll use with your metamodels so that there are full paths from the most abstract metamodels to the metamodels of all of your target platforms.
5. Define the annotation models that these mapping techniques require.
6. Implement your mapping techniques either by using tool support or by describing the steps necessary to manually carry out the techniques.
7. Conduct iterations of the project. Each iteration will add detail to one or more models describing the system at one or more levels of abstraction. You'll map these additional details all the way down to the target platforms.

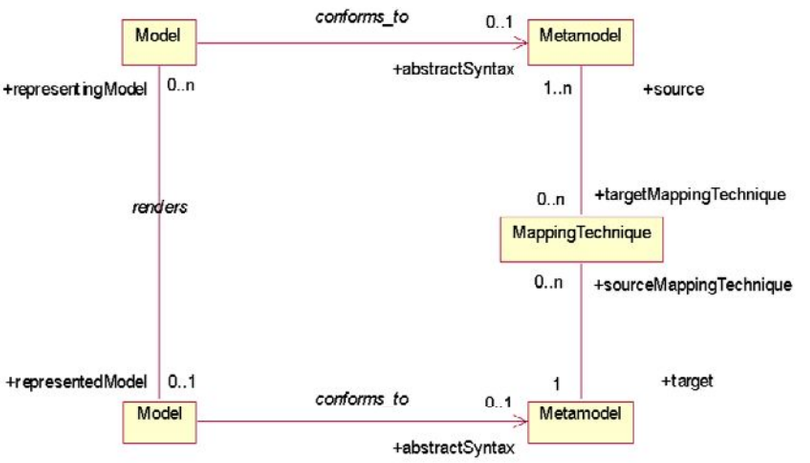


Fig. 6. Model Representation

9 Accelerated MDA

Figure 6 shows instances of PIM and PSM models and metamodels and how these instances relate to one another. These models can themselves be populated with instances. The PSM model instance contains all of the semantic information of the original PIM model instance, as well as all of the information added as a result of the mapping technique. Because the PSM metamodel describes a platform, the PSM also includes – at least by reference – all of the elements that comprise the platform.

However, the PSM may *not* be the bottom of the platform stack. The PSM can itself be modeled as a PIM, which in turn is mapped to another PSM via a different mapping technique. The resulting PSM, which we'll call PSM', now contains all of the semantic information of the original PIM model instance, the information added as a result of the application of the first mapping technique, *and* the information added by the application of the second mapping technique. Because each PSM metamodel describes a platform, the second-level PSM also includes the elements that comprise *both* platforms.The recursion ends when the last PSM (a PSM with some number of primes) is directly implementable, either by generating code or by being directly executable on the virtual machine modeled by the last PSM. The platform stack, and the distinction between the last platform (a PrimitiveRealization) and those platforms above it, are illustrated in Figure 3.

This approach is ultimately flexible. Any model, regardless of exactly how platform-independent it is, can have a metamodel defined for it, and developers can easily model applications using their favorite notations, conceptually optimized for the platform types to which the models are to be mapped. However, this approach requires the definition of multiple mapping techniques to preserve semantics at each level. The result is that platforms can become "silos" in which certain mappings can only be applied under a certain set of assumptions. For example, once a particular kind of inheritance has been applied in one metamodel, it will be difficult to use a different kind of inheritance in a subsequent model.

An alternative is to use the *same* metamodel to capture each platform. In this formulation, a model is always expressed using the same concepts, enough to execute the model. In addition, the modeler chooses the concepts such that the model can be translated into any arbitrary platform. This "Executable and Translatable UML" doesn't contain any information about software structure, only about behavior of the subject matter under study. The metamodel describing Executable and Translatable UML is therefore much smaller than UML.

When an Executable and Translatable UML model is rendered as an implementation, it is annotated as described above and woven together with the platform specifics by a mapping technique that is completely general – it needs only be written once. Platform-specific mappings, of course, will vary, but that's inherent in the fact that platforms differ.

This accelerated approach bypasses the PSM. There's no need to capture the conjoining of the developer model and the platform as a distinct model. The developer model and the platform model are woven together directly into an implementation. This approach also increases the applicability of tools, because there's only one

metamodel on which to operate, rather than multiple metamodels (one for each platform).

10 Conclusion

MDA is the OMG□s next step in solving integration problems through open, vendor-neutral interoperability specifications. It is constantly evolving based on the experiences of OMG members in creating standards for implementation language-independent models in CORBA and developing standards such as UML and MOF. The key concepts described in this paper form the foundation for ongoing exploration of what MDA will be, while the process outline provides a healthy start toward the definition of how software development teams can maximize the benefits of MDA.
check (mail orders only).

Model-Based Development of Embedded Systems*

B. Schätz[1], A. Pretschner[1], F. Huber[2], and J. Philipps[2]

[1] Institut für Informatik, Technische Universität München
Arcisstr. 21, 80290 München, Germany
[2] Validas Model Validation AG
Software-Campus, Hanauerstr. 14b, 80992 München, Germany

Abstract. Model-based development relies on the use of explicit models to describe development activities and products. Among other things, the explicit existence of process and product models allows the definition and use of complex development steps that are correct by design, the generation of proof obligations for a given transformation, requirements tracing, and documentation of the process. Our understanding of model-based development in the context of embedded systems is exposed. We argue that the concept of model-based development is orthogonal to a specific process, be it agile or rigorous.

1 Introduction

Intuitively, model-based development means to use diagrams instead of code: Class or ER diagrams are used for data modeling, Statecharts or SDL process diagrams abstractly specify behavior. CASE tool vendors often praise their tools to be model-based, by which they mean that their tools are equipped with graphical editors and with generators for code skeletons, for simulation code, or even for production code.

However, we do not believe that model-based development should be regarded as the application of "graphical domain-specific languages". Instead, we see model-based development as a paradigm for system development that besides the use of domain-specific languages includes *explicit* and *operational* descriptions of the relevant entities that occur during development in terms of both product and process. These descriptions are captured in dedicated models:

Process models allow the description of development activities. Because of the explicit description, activities are *repeatable*, *undoable* and *traceable*. Activities include low-level tasks like renamings and refactorings, but also higher-level domain-specific tasks like the deployment of abstract controller functionalities on a concrete target platform.

* This work was in part supported by the DFG (projects KONDISK/IMMA, InOpSys, and Inkrea under reference numbers Be 1055/7-3, Br 887/16-1, and Br 887/14-1) and the DLR (project MOBASIS).

J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 298–311, 2002.

Product models contain the entities that are used for the description of the artifact under development and the necessary parts of its environment, as well as the relations between these entities. All activities in the process models are defined in terms of the entities in the product models.

We believe that many important problems in industry like the coupling of different tools for different development aspects (e.g., data aspects, behavior aspects, scheduling and resource management aspects) are still unsolved because of a lack of an underlying coherent metaphor. We see explicit product and process models as a remedy to this problem.

Overview. In this paper, we provide a rather abstract treatment of our understanding of model-based development. An extended version of this paper has been published as a technical report [11]. As application domain, we choose that of embedded systems, but the general ideas apply to other domains as well. The article's remainder is organized as follows. We kick off with the basic idea of explicit process and product models in Section 2. The essence of product and process models is described in Sections 3 and 4, respectively. In Section 4, we argue that model-based development may be used in different processes, agile or rigorous. Related work is presented in Section 5, and Section 6 concludes.

2 Models

The shift from assembler towards higher languages like C or Ada essentially reduces to the incorporation of abstractions for control flow (like alternative, repetition, exceptions), data descriptions (record or variant types), and program structure (modules) into these higher languages. Middleware (like CORBA, .NET) are further examples of increasingly abstract development. We consider model-based development to be a further step in this direction. It aims at higher levels of domain-specific abstractions as seen, at a low level, in the abstraction step performed in `lex`. In the field of embedded controllers, the concepts of capsules and connectors of, e.g., the UML-RT are used as well as state machines to describe component behavior. That these abstractions have intuitive graphical descriptions is helpful for acceptance, but not essential for the model concept. Furthermore, in model-based development there is no need to exclusively rely on one particular description technique, or rather the underlying concept.

What are the advantages of model-based development? One advantage is independence of a target language: Models can be translated into different languages like C or Ada for implementation. For graphical simulation, other languages are likely better suited. Again, this is in analogy with the abstraction step, or, inversely, compilation of programming languages: C code can be translated into a number of different assembler languages.

The key advantage, however, is that the product model, which subsumes the abstract syntax of a modeling language, restricts the "degrees of freedom" of design in comparison with programming languages. This is akin to modern programming languages that restrict the degrees of freedom of assembler languages

by enforcing standard schemes for procedure calls, procedure parameters and control flow. In a similar sense, Java restricts C^{++} by disallowing, among other things, multiple inheritance. Ada subsets like Ravenscar or SPARK explicitly restrict the power of the language, e.g., in terms of tasks. The reason is that these concepts have proved to yield artifacts that are difficult to master.

Model-based development incorporates the aspects of *abstraction* and *restriction* in high level languages. This happens not only at the level of the product but also at the level of the process. Working with possibly executable models not only aims at a better understanding and documentation of requirements, functionality, and design decisions. Models may also be used for generating simulation and production code as well as test cases. We consider the integration of different models at possibly different levels of abstraction as the key to higher quality and efficiency of the process we propose. Integration is concerned with both products and processes, on a horizontal as well as a vertical level.

Horizontally, different aspects have to be integrated. These aspects reflect a separation of concerns by means of abstractions. They deal with concepts like structure, functionality, communication, data types, time, and scheduling. Structural abstractions concern logical as well as technical architectures, and their relationship. Functional abstractions discard details of the actually desired behavior of the system. Communication abstractions allow the developer to postpone decisions for, e.g., hand-shaking and fire-and-forget communications. Data abstractions introduce data types at a level of granularity that increases over time. and helps in building functional, communication, and structural abstractions. Timing and scheduling abstractions enable the developer to neglect the actual scheduling of components—or even abstract away from timing by relying solely on causality—in early development phases. Other aspects like security, fault tolerance, or quality-of-service may be considered as well. While these aspects are not entirely orthogonal one from another, thinking in these terms allows a better structuring of systems.

Vertically, different levels of abstraction[1] for each of the above aspects have to be brought together in a consistent manner. This applies to both integrating different structural abstractions and integrating structure with functionality and communication. Furthermore, different levels of abstractions in all areas have to be interrelated: Refinements of the black box structure have to be documented and validated, and the same is obviously true for functional and data refinements. Since in a sense, possibly informal requirements also constitute abstractions, tool supported requirements tracing is a must for such a model-based process.

In an incremental development process, increments (or parts of a product) have to be integrated over time (Figure 1). While this figure suggests that the concepts of level of abstraction and increments are orthogonal, one might well argue that

[1] Note that the term *abstraction* is used in an ambiguous manner: abstractions in the mathematical sense and abstractions on a conceptual level where constructs for describing one view of a system are considered (i.e., ontological entities).

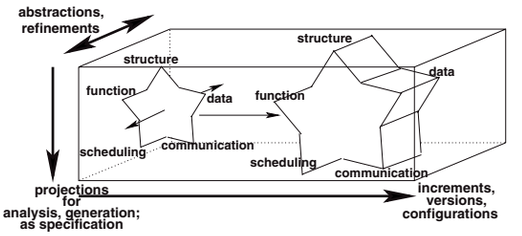


Fig. 1. Model-based development

a refinement step does constitute an increment. The reason for the distinction is that abstractions and refinements form special increments the correctness of which might, in a few cases, be proved or automatically tested.

Process and Product Models. In the UML, the notion of a model is used to describe the elements and concepts used during the development process, e.g. class, state, or event. Since, however, this distinction is too coarse for the description of the model-based approach, here more fine-grained notions of models will be used: *process, product, conceptual,* and *system* models. In the following, these models are explained in more detail and related to each other. We use the domain of embedded systems development and the CASE tool AutoFocus [7] with its UML-RT-like description techniques for illustration.

The first two models are used to describe the development process from the engineer's and thus the domain model point of view. Together process and product models form the *domain* model:

Process Model: The process model consists of the description of *activities of a development process* and their relations. In the domain of embedded reactive systems, e.g., the process model typically contains modeling activities ("define system interface", "refine behavior") as well as activities ("generate scenarios or test cases", "check refinement relation", "compute upper bound for worst case execution time"). The activities are related by their dependency between them defining a possible course of activities throughout the development process. By relating them to a product model, process patterns can be formalized as activities and thus integrated in the process model.

Product Model: The product model consists of the description of those aspects of a system under development *explicitly* dealt with during the development process and handled by the development tool. For embedded systems, a product model typically contains domain concepts like "component", "state" or "message", as well as relations between these concepts like "is a port of a component", etc. In addition to these more conceptual elements, used for the description of the product, more semantically oriented concepts like "execution trace" are defined to support, for example, the simulation

of specification during the development process. Finally, it contains process oriented product concepts like “scenario” or “test case”, supporting the definition of process activities.

3 Product

The product model describes the aspects, concepts and their relations needed to construct a product during the development process. Thus, it supplies the ‘language’ to describe a product. Usually, this language is represented using view based description techniques like structural, state oriented, interaction oriented, or data-oriented notations. The concrete product itself is an instance of the product model, for example represented by system structure diagrams, state transition diagrams, or MSC-like event traces.

Since process activities are defined as changes of instances of the product model, a process model can only be defined on top of a product model. The granularity of a product model also defines the expressiveness and thus the quality of the process model. Using both models, detailed development processes can be described, accessible to CASE support.

3.1 Structure of the Product Model

While the ‘abstract syntax’ is sufficient to describe conceptual relations of abstract views or the functionality of modeling activities during the process, a semantical relation is needed to define or verify more complex semantical dependencies of views as well as properties of activities (like refining activities or activities not changing the behavior as, e.g., refactoring). Since the semantical and the conceptual part of the product model are used differently in the model-based approach, the product model is broken up into two sub models:

Conceptual Model: The conceptual model consists of the *modeling concepts* and their relations used by the engineer during the development process. The conceptual model is independent of its concrete syntactic representation used during the development process. Typical domain elements for embedded systems are concepts like “component”, “port”, “channel”, “state”, “transition”, etc. Typical relations are “is_port_of”, “is_behavior_of”, etc. Figure 2 shows a simplified part of the AutoFocus conceptual model. Besides those low-level concepts, concepts like “requirement” or “test case” including relations like “discharged_by” or “is_test_case_of” are included.

System Model: The system, or semantical, model consists of *semantical concepts* needed to describe the system under development. A typical element is “execution sequence”. Typical relations are “behavioral refinement” or “temporal refinement”.

As shown in Figure 3, the notion of the conceptual model is closely related to the notion of views and description techniques. Views of a product correspond

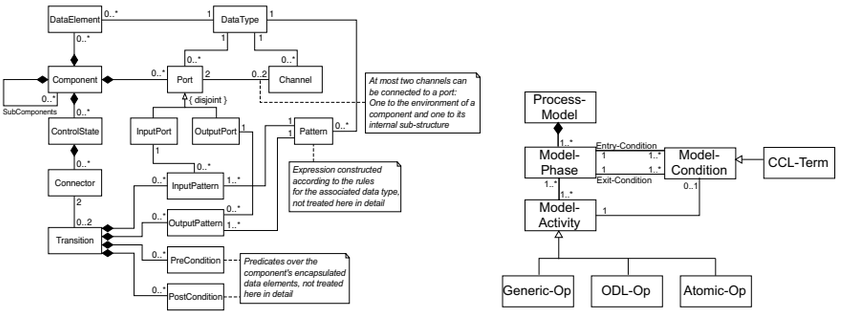


Fig. 2. Simplified conceptual product model (left) and process model (right)

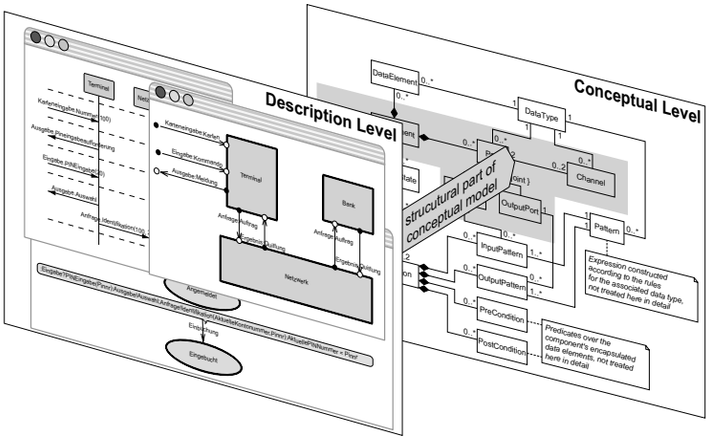


Fig. 3. Models and views

to abstractions of an instance of the conceptual model (e.g., horizontally: structure, communication; vertically: component, subcomponent) and are represented using description techniques.

A semantic interpretation is assigned to the instances of the conceptual model by instantiating the system model according to its relation to the conceptual model. In this way, for instance, refinement relations can be proved, or the validity of timing constraints can be checked.

3.2 Application of Models

The purpose of the product model is to support a more efficient and sound development process by providing a domain-specific level of development. For the engineering process, the model is used *transparently* through views of the model

in form of description techniques as described above and interaction mechanisms supporting the development process. Two mechanisms can be used: *consistency conditions* and the *definition of process activities*. Consistency conditions exist at three different levels:

Invariant Conceptual Consistency Conditions: They are expressible within the conceptual model, and hold invariantly throughout the development process. Therefore, they are enforced during construction of instances of the conceptual model. Since generally these are only simple consistency conditions, they can be defined as multiplicities of relations of the conceptual model. Examples are syntactic consistency conditions as used in AutoFocus like "a port used during the interaction of a component is part of its interface" or "a channel and its adjacent ports have the same types".

Variant Conceptual Consistency Conditions: Like the invariant conceptual conditions, these conditions can be expressed completely within the conceptual model. However, unlike those, they may be relaxed during certain steps of the development process and are enforced during others. Examples are methodical consistency conditions like "The dependency graphs of variable assignments are non-circular", or "All transitions leaving a state have disjoint patterns thus ensuring deterministic behavior".

Semantic Consistency Conditions: These conditions are not expressible in the conceptual model. Since, generally, they cannot simply be enforced, the validity of these conditions is not guaranteed throughout the development process but must be checked at defined steps of the process. Examples are semantic conditions: "The glass box behavior of a component refines the black box behavior", "The behavior of an event trace of a component is a refinement of the behavior of the component", or "The timing behavior of a component respects its worst case time bounds".

In general, the distinction between invariant and variant conceptual consistency conditions is a matter of flexibility and rigorousness of the development process supported by the underlying model. In the AutoFocus approach we use CCL (Consistency Constraint Language, [7], [12]) to define conceptual consistency conditions. Similar to the OCL, it corresponds to a first order typed predicate calculus with the types (classes) and relations (associations) of the conceptual model; expressions are evaluated using an instance of the conceptual model as universe. AutoFocus offers an evaluation mechanism for CCL expressions returning all counterexamples of the current instance of the conceptual model.

Variant conceptual consistency conditions as well as generic primitive operations of the conceptual model (introducing/removing instances of elements and relations) provide the base operations needed to access the conceptual model from the process point of view. Together with the semantic operations like checking semantical consistency conditions they form the basic activities of a development process as explained in Section 4.

4 Process

As mentioned above, the justification of the product model is its application in the definition of a process model. By the use of a detailed product model we can (1) give a detailed definition of the notions of *phase* and *activity* in terms of how they interact with the conceptual model, (2) increase the soundness of the development process by introducing semantical consistency conditions or sound activities with respect to the system model, and (3) most importantly, add CASE support to the process to increase efficiency of development.

4.1 Structure of the Process Model

As shown in Figure 2, a simplified process model consists of

Phases: Phases define a coarse structure of a process. They can be associated to conditions that must be satisfied before or after the phase. Each phase has an associated set of activities that can be performed during this phase. Typical examples are phases like "Requirements analysis" or "Module implementation". Simplified examples for corresponding conditions are "Each requirement must be mapped to an element of the domain model" to hold at the end of requirements analysis or "Each component has an implementable time-triggered behavior" to hold at the end of the implementation phase. Using the consistency mechanism, development is guided by checking which conditions must be satisfied to move on in the process.

Activities: In contrast to the unstructured character of a phase, an activity is an operationally defined interaction with an instance of the conceptual model and thus executable. An activity can be extended with a condition stating its applicability at the current stage for user guidance. An activity is either a generic operation generated from the conceptual model, an atomic operation supplied by the system model, for example checking semantic consistency, or a complex operation constructed from the basic operations.

In more complex processes, phases and activities usually consist of sub-phases and sub-activities, respectively. Since phases and activities are defined in terms of the product model, their dependencies can be expressed in terms of the product rather as in generally unspecific ways as found in general process description languages [3].

Examples for basic operations include simple construction steps like "generation of a new state of a component" or "introduction of a new transition into the state-description of a component". Complex operations include refactoring steps like "pull up a subcomponent out of its super-component to become a component of the same level (involving a change in the subcomponent relation and a relocation of ports and channels)".

Process activities generally consist of a collection of simple and complex operations to be applied during an activity. Complex operations can be defined in the form of extended pre/post-conditions, describing a transformation of instances

of the conceptual model. In the AutoFocus approach these operations can be defined in ODL (Operation Definition Language, [12]), an extension to the OCL-like CCL introduced above. ODL allows to precisely define the pre- and postconditions (including user interaction) of an operation in terms of the instance of a conceptual model. ODL definitions are executable but come in a logical form that supports the verification of conceptual properties of the operation. These properties include stability w.r.t. consistency conditions or semantical properties like behavioral refinement.

To illustrate how process, conceptual and system models interact, we use the example of the "elimination of dead states" refactoring step that reduces code size. In this example, a control state including all its adjacent transitions is removed from an automaton provided this state is marked as unreachable.

Process: On the level of the process model, an activity "Show unreachability of a state" must be introduced. This step may either be a single atomic operation (and can be carried out, e.g., by some form of model checking algorithm) or a more complex operation requiring user interaction. These operations correspond to operational relations of the system model. Furthermore, the activity "Remove dead state" removes all transitions leading to a state marked unreachable as well as the state.

Conceptual: On the level of the conceptual model, the concept of "unreachability" of a state, e.g. as a state annotation, is introduced. If no user interaction is required for the proof of unreachability, this extension is sufficient. Otherwise conceptual elements of proof steps must be added, as well as additional concepts like state/assignment of variable, or precondition of a transition

System: On the level of the system model the semantics have a direct effect: in case of an atomic operation, there is an operational notion of the semantical predicate "unreachability", e.g., in form of a model-checking algorithm. This operation adds the conceptual annotation "unreachable" to a state unreachable according to the semantics of the system model. In case of an operation requiring user interaction, the system model is used via atomic operations corresponding to operational relations of the system model (e.g. combining parts of a proof, applying modus ponens). Besides this application of the system model "at run-time of the CASE tool", the system model is also applied "at build-time" to prove the correctness of the refactoring step.

Since an activity as the atom of a process describes how a product is changed, an activity can be understood as a process pattern in the small. Additionally, each activity is described in an operational manner. Furthermore, by the use of the system model, properties like "soundness considering behavioral equivalence" or "executability of the specification" can be established for activities and phases. This combination of user guidance by consistency conditions, of executable activities, and of the possibility for both arbitrary and provably sound process activities and states of a product, is directed at improving the efficiency of the CASE based development process.

4.2 Agility

Model-based development is orthogonal to the degree of rigorosity of a process. Without giving a clear definition of agile processes, we illustrate this claim by embedding aspects of Extreme Programming [1] into model-based development.

Language: While generally associated with Java, XP itself does not prescribe any particular language, and we might thus use any model-based formalism appropriate for a given application domain. Furthermore, we do not exclude classical languages from model-based development (in the context of aspect oriented programming we will, however, argue that general purpose languages are difficult to handle in terms of correctness). XP clearly is code centered. However, since we advocate the use of operational models not only for documentation purposes, in this context there is no qualitative difference between using a low level and a high level language. One of our main points, the restriction of general purpose languages, is orthogonal to this aspect.

Testing: One of the core ideas of XP is to continually test the system under development. In fact, test cases are the only formal means of specification. Incorporating explicit process activities for testing and implementation steps does by no means contradict the principles of XP nor those of model-based development.

Refactoring: The activity of refactoring [4] is a common technique in incremental processes like XP. Refactoring is the activity of restructuring an artifact (code) without altering its behavior. Within the model-based approach, these refactoring steps may well be incorporated in the process model as explicit process activities as well.

Augmenting agile approaches like XP by model-based constructs is likely to pay off. Firstly, if one accepts domain specific languages to be a good choice, then it is a good choice for agile methods as well. Remember that the explicit existence of product and process models does not mean they are visible to the user. Rather, the contrary is desirable. Secondly, every automatic approach to test case generation clearly requires a clear and explicit understanding of the semantics of the language that is used. Thirdly, refactoring relies on behavioral equivalence, and behavior is always dependent on an observer: is execution time part of the behavior or not? An explicit semantics must thus complement the intuitive notion when building, for instance, safety critical systems. The definition or automatic generation of test cases for checking behavioral equivalence of the system with its refactored counterpart obviously requires this semantics as well. Assigning explicit meaning to the language the XP engineers deploy would thus result in an explicit system model which could be linked to the refactoring patterns in the process model.

Intuitively, model-based development seems to imply rather rigorous processes. However, depending on the rigor of the defined process activities, it can support very flexible processes as well. Thus, applying the model-based approach to XP, this could result in a less code-centered but still flexible ‘Extreme Modeling’ approach.

5 Related Work

Though widely used, we are not aware of explicit definitions of "model-based development". Especially, this term is often used in a more restricted sense, e.g. domain oriented software architectures [15]. Harel [5] is concerned with using statecharts for behavior specification. The approach presented by Sgroi et el. [13] and Keutzer et al. [8] is similar to ours, in terms of the incorporation of different levels of abstraction, separation of concerns—in particular, computation and communication—, and the emphasis on explicit system models. The especially CASE relevant distinction of models within a layered approach as we propose it complements their line.

While clearly code centered, aspect oriented programming [9]—or, more generally, separation of concerns—is similar in its vision w.r.t. finding ontological entities, i.e., abstractions, for aspects like concurrency, exception handling, etc. Differing from our approach, the idea is to incorporate these abstractions into general purpose languages like Java or C rather than to use dedicated domain specific languages. While there are static analysis tools for these languages, their expressive power renders these analyses most difficult—this is one reason why we emphasize the *restriction* of existing languages. In its pure form, AOP does not require an explicit product nor process model.

The notion of explicit product and process models is also found in the area of Process Definition Languages [3], however, focusing on user participation and neglecting the importance of a domain-specific, detailed product model to define a process upon.

In the UML, a detailed model of the product is defined, integrating different views of a product. However, semantical relations exceeding the structural relations of the conceptual model are missing. Since UML is focused on the conceptual product model, it must be related to development activities. While the RUP defines a process on top of the UML, it does not make use of the fine grained meta model underlying those description techniques. Activities of the process, their preconditions and results are not defined in terms of the UML meta model; rather, the RUP outlines the phases to be carried out and suggests the description techniques that are useful for each phase.

Modeling approaches like MOF (Meta Object Facility) rather focus on technical aspects of how to implement and access models and meta models, but do not address their application in defining domain-specific development processes. The OMG's model driven architecture [14] aims at the definiton of platform-independent models in a platform-independent language (UML) that are later mapped to platform-specific models (CORBA, SOAP, etc.). It is thus concerned with the aspect of communication as well as structure as described in Section 2, however focusing on the architecture of a product.

Graphical editors in tools like Together or ArgoUML [10], for instance, concentrate on an explicit (UML) meta model but do not take into account a process model. Development platforms like Eclipse[2], the latest in IDE development, de-

[2] http://www.eclipse.org

fine process patterns (e.g., refactorings) but do not do this in an explicit manner. As far as we know, there is no explicit product model, either.

6 Conclusion

Our vision of model-based development rests on two pillars: Explicit *product models*, which for the developer appear as domain-specific languages, and explicit *process models*, which define the developer's activities that transform early, abstract, partial products to the final, concrete and complete products that are ready to be delivered and deployed.

The *benefits of model-based development* come from the *interaction of process and product models* and their *realization in a CASE tool*: Firstly, *complex design steps* such as refactorings or the introduction of complex communication patterns [11] between components can be naturally defined and performed in a tool. Secondly, the application of such design steps naturally leads to a *development history* that can be recorded in the tool and used for a kind of high-level configuration and version management. Finally, the *requirements and design rationales* that influence design steps can be *traced and documented* throughout the complete development process.

The main goal of model-based development is to produce high-quality software at acceptable cost. While high software quality can be achieved even now—as demonstrated by avionics software—, cost and development time are usually forbidding. Model-based development aims at improving not only the *product*, but also the *process* that leads to the product, making high-quality software development more affordable. In particular, it aims increasing the efficiency of the development not only of single products but of related product families.

However, model-based development is not without risk. It is not obviously clear whether a seamless development process from early design to final target code is feasible: Some design steps might demand knowledge of environment properties which are difficult to formalize. Design steps in the later phases will require precise knowledge of the target platform, for instance to access device drivers or in order to estimate the worst case execution times which are needed as input for scheduling algorithms. Even if this knowledge is formalized and incorporated into the product model—as, for example, partly done in the Giotto language [6]—, more pragmatic problems, like the integration of legacy code, tailoring to customer-specific coding and certification standards or possibly just idiosyncrasies in compiler or operating system technologies can hamper our ideal of a seamless process.

These problems can—with varying degrees of difficulty—be solved. The main problem is that in contrast with, for instance, compiler construction, they can be solved not by tool builders alone, but only in close cooperation with domain experts: A model-based development will necessarily be domain-specific. Finding common vocabularies and notations to define the conceptual, system and product models is a rather ambitious goal.

Still, in view of our experiences with the AutoFocus project we are optimistic that such tools can be built. Although we do not yet have enough experience with industrial-size projects, we obtained satisfying results with some core aspects of such systems (deployment of systems on 4-bit and 8-bit microprocessors, schematic introduction of security aspects, custom scheduling algorithms to distribute computation effort over time). That the close integration of domain properties into CASE tools is feasible has been demonstrated, for example, for simultaneous engineering in process automation [2].

References

1. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 1999. 307
2. K. Bender, M. Broy, I. Péter, A. Pretschner, and T. Stauner. Model based development of hybrid systems: specification, simulation, test case generation. In *Modelling, Analysis and Design of Hybrid Systems*, LNCIS. Springer, 2002. To appear. 310
3. Jean-Claude Derniame, Badara Ali Kaba, and David Wastell, editors. *Software Process: Principles, Methodology and Technology*. Springer, 1999. LNCS 1500. 305, 308
4. M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley, 1999. 307
5. D. Harel. Biting the silver bullet: Toward a brighter future for system development. *IEEE Computer*, 25(1), January 1992. 308
6. Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A time-triggered language for embedded programming. In *Proceedings of EMSOFT 2001, LNCS 2211*, 2001. 309
7. Franz Huber and Bernhard Schätz. Integrated Development of Embedded Systems with AutoFocus. Technical Report TUMI-0701, Fakultät für Informatik, TU München, 2001. 301, 304
8. K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 19(12), December 2000. 308
9. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. European Conference on Object-Oriented Programming (ECOOP'97)*, Springer LNCS 1241, 1997. 308
10. J. Robbins. *Cognitive Support Features for Software Development*. PhD thesis, University of California, Irvine, 1999. 308
11. B. Schätz, A. Pretschner, F. Huber, and J. Philipps. Model-based development. Technical Report TUM-I0204, Institut für Informatik, Technische Universität München, 2002. 299, 309
12. Bernhard Schätz. The ODL Operation Definition Language and the AutoFocus/Quest Application Framework AQuA. Technical Report TUMI-1101, Fakultät für Informatik, TU München, 2001. 304, 306
13. M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal Models for Embeded System Design. *IEEE Design& Test of Computers, Special Issue on System Design*, pages 2–15, June 2000.

14. R. Soley. Model Driven Architecture. OMG white paper, 2000. 308 308
15. James Withey. Implementing model based software engineering in your organization: An approach to domain engineering. Technical Report CMU/SEI-94-TR-0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1994. 308

Author Index