Preproceedings of the Workshop on

Software Evolution through Transformations: Model-based vs. Implementation-level Solutions

— SETra 2004 —

A satellite event of ICGT 2004
October 2nd 2004, Rome, Italy

Organized by Reiko Heckel[1] and Tom Mens[2]

[1] *Universität Paderborn, Germany,* reiko@upb.de

[2] *Université de Mons-Hainaut, Belgium,* tommens@vub.ac.be

Supported by

The European Science Foundation (ESF) Network RELEASE:
Research Links to Explore and Advance Software Evolution

and

The European Research Training Network SegraVis:
Syntactic and Semantic Integration of Visual Modeling Techniques
www.segravis.org

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Preface

Motivation and Objectives

Changes to software artifacts and related entities tend to be progressive and incremental, driven, for example, by feedback from users and other stakeholders, such as bug reports and requests for new features, or more generally, changes of functional or non-functional requirements. In general, evolutionary characteristics are inescapable when the problem to be solved or the application to be addressed belongs to the real world.

There are different strategies to address evolution. Model-based software development using the UML, as proposed by the OMG's MDA initiative, addresses evolution by automating (via several intermediate levels) the transformation of platform-independent design models into code. In this way, software can be evolved at the model level without regard for technological aspects, relying on automated transformations to keep in sync implementations on different target platforms.

Classical re-engineering technology, instead, starts at the level of programs which, due to the absence or poor quality of models, provide the only definite representation of a legacy system. The abstractions derived from the source code of these systems are not typically UML models, but they may play a similar role in the subsequent forward engineering steps.

Which approach is preferable in which situation, and how both strategies could be combined, is an open question. To answer that question and to implement any solutions deriving from these answers we require

- a *uniform understanding* of software evolution phenomena at the level of both models and programs, as well as of their interrelation;
- a *common technology basis* that is able to realize the manipulation of artifacts at the different levels, and to build bridges between them.

It is the hypothesis of this workshop that *graphs*, seen as conceptual models as well as data structures, defined by meta models or other means, and *transformations*, given as program or model transformations on graph- or tree-based presentations, provide us with unifying models for both purposes.

Transformations provide a very general approach to the evolution of software systems. Literally all activities that lead to the creation or modification of documents have a transformational aspect, i.e., they change a given structure into a new one according to pre-defined rules. As a common representation of artifacts like models, schemata, data, program code, or software architectures,

graphs have been used both for the integration of development tools and as a conceptual basis to reason on the development process and its products.

Based on this conceptual and technological unification, it is the objective of the workshop to provide a forum for studying software evolution phenomena and discussing their support at different levels of abstraction.

Topics of interest include

- graph-based models for analysis, visualization, and re-engineering
- software refactoring and architectural reconfiguration
- model-driven architecture and model transformations
- consistency management and co-evolution
- relation and tradeoffs between model- and program-based evolution

Acknowledgements

Due to support by the *European Science Foundation (ESF)* through the Scientific Network RELEASE (Research Links to Explore and Advance Software Evolution) the workshop is free of participation fees. It is the official kick-off meeting of the Working Group on Software Evolution of the *European Research Consortium for Informatics and Mathematics (ERCIM)* and a meeting of the *European Research Training Network* SegraVis on Syntactic and Semantic Integration of Visual Modeling Techniques.

Reiko Heckel, University of Paderborn, Germany
Tom Mens, Université de Mons-Hainaut, Belgium

Program Committee

- Giulio Antoniol (Università degli Studi del Sannio, Italy)
- Andrea Corradini (Università di Pisa, Italy)
- Stephane Ducasse (University of Bern, Switzerland)
- Jean-Marie Favre (Institut d'Informatique et Mathématiques Appliquées de Grenoble, France)
- José Fiadeiro (University of Leicester, United Kingdom)
- Harald Gall (Technical University of Vienna, Austria)
- Martin Große-Rhode (Fraunhofer ISST Berlin, Germany)
- Reiko Heckel (Universität Paderborn, Germany) [co-chair]
- Anton Jansen (University of Groningen, The Netherlands)
- Dirk Janssens (University of Antwerp, Belgium)
- Juan F. Ramil (Open University, United Kingdom)
- Ashley McNeile (Metamaxim Ltd., London, United Kingdom)
- Tom Mens (Vrije Universiteit Brussel, Belgium) [co-chair]

Workshop Program

The workshop offers presentations of different lengths.

L: long presentation (15 min of lecture + 5 min discussion)

S: short presentation (5 min of lecture + 5 min discussion)

Saturday, October 2nd

8.45-9.00: Opening and Welcome

9.00 - 10.00 Transformation Technology

L: *Leveraging UML Profiles to generate Plugins from Visual Model Transformations*, Hans Schippers, Pieter Van Gorp, Dirk Janssens **p. 7**

S: *The Side Transformation Pattern - making transforms modular and reusable*, Edward D. Willink, Philip J. Harris **p. 18**

S: *An Algebraic Baseline for Model Transformations in MDA*, Artur Boronat, Josá Á. Cará, Isidro Ramos **p. 30**

S: *Evolution of Language Interpreters*, Ralf Laemmel **p. 46**

S: *Reflective designs*, Robert Hirschfeld, Ralf Laemmel **p. 52**

10.00 - 10.30 Meta Models I

L: *Towards a Megamodel to Model Software Evolution Through Transformations*, Tam NGuyen, Jean-Marie Favre **p. 56**

S: *Modeling Software Evolution by Treating History as a First Class Entity*, Stephane Ducasse, Jean-Marie Favre, Tudor Girba **p. 71**

10.30 - 11.00 Coffee

11.00 - 11.30 Meta Models II

L: *Towards Integrating CVS Repositories, Bug Reporting and Source Code Meta-Models* , Giuliano Antoniol, Massimiliano Di Penta, Harald Gall, Martin Pinzger **p. 83**

S: *Behavioral Refinement of Graph Transformation-Based Models*, Sebastian Thöne, Reiko Heckel **p. 95**

11.30 - 12.30 Analysis

L: *Detecting Structural Refactoring Conflicts using Critical Pair Analysis* , Tom Mens, Gabriele Taentzer, Olga Runge **p. 105**

L: *Predicting Incompatibility of Transformations in Model-driven Development*, Mehdi Jazayeri, Johann Oberleitner **p. 120**

S: *Proof Transformation via Interpretation Functions*, Piotr Kosiuczenko **p. 128**

S: *On the Evolution Complexity of Design Patterns*, Tom Mens, Amnon H. Eden **p. 136**

12.30 - 14.00 Lunch

14.00 - 15.00 Architectural Evolution

L: *Evolution Through Architectural Reconciliation*, Paris Avgeriou, Nicolas Guelfi, Gilles Perrouin **p. 152**

L: *Towards an Integrated View on Architecture and its Evolution*, Martin Pinzger, Michael Fischer, Harald Gall **p. 168**

S: *Fresco: Flexible and Reliable Evolution System for Components*, Yves Vandewoude, Yolande Berbers **p. 182**

S: *Dynamic software assembly for automatic deployment-oriented adaptation*, Anthony Savidis **p. 191**

15.00 - 15.30 Discussion

15.30 - 16.00 Coffee

16.00 - 18.00 ERCIM and ESF Meeting

During this meeting we will discuss the details of the new ERCIM Working Group on Software Evolution we are currently starting up. We will also discuss some administrative matters for the ESF RELEASE network. All ERCIM and RELEASE members are expected to join this meeting, but the meeting is also open to anyone else who is potentially interested.

Leveraging UML Profiles to generate Plugins from Visual Model Transformations

Hans Schippers*, Pieter Van Gorp, Dirk Janssens

Formal Techniques in Software Engineering
Universiteit Antwerpen, Belgium
{hans.schippers,pieter.vangorp,dirk.janssens}@ua.ac.be
Aspirant of the Fund for Scientific Research (FWO) - Vlaanderen

Abstract

Model transformation is a fundamental technology in the MDA. Therefore, model transformations should be treated as first class entities, that is, models. One could use the metamodel of SDM, a graph based object transformation language, as the metamodel of such transformation models. However, there are two problems associated with this. First, SDM has a non-standardized metamodel, meaning a specific tool (Fujaba) would be needed to write transformation specifications. Secondly, due to assumptions of the code generator, the transformations could only be deployed on the Fujaba tool itself. In this paper, we describe how these issues have been overcome through the development of a template based code generator that translates instances of a UML profile for SDM to complete model transformation code that complies to the JMI standard. We have validated this approach by specifying a simple visual refactoring in one UML tool and deploying the generated plugin on another UML tool.

Key words: Refactoring, Model Transformation, SDM, JMI

1 Introduction

As Sendall and Kozaczynski state [SK03], model transformation can be seen as the *heart and soul* of model driven software development. In terms of OMG's Model Driven Architecture (MDA [Obj01]), PIM-to-PSM transformations come to mind immediately, but that is only half the story. Indeed, beside these *refinements* (a special kind of *translations*), there is another important class of model transformations: *rephrasings* [Gor04]. These are transformations within the same metamodel (intra-metamodel), which could be applied to change a model because of evolving requirements, or to enhance a model's internal structure without modifying its external behavior (refactoring). Recent experiments [GEJ03] have shown that Fujaba's Story Driven Modeling (SDM [FNTZ98]) can be used as a language for developing transformations of this class.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

However, one was restricted to Fujaba as its development environment for two reasons. The first has to do with the fact that SDM is an independent, non-standard metamodel, and is only implicitly present in the Fujaba source code. Therefore, SDM specifications could only be written with the Fujaba editor. The second problem is that model transformations developed with Fujaba can only be deployed on the Fujaba repository itself. Its cause is that the Fujaba code generator only integrates with code complying to a Fujaba proprietary API. More specifically, it generates code that is based on a specific association framework. Obviously, these issues stand in the way of the approach becoming mainstream. They have been overcome by, on the one hand, designing a UML profile for SDM, implying that any CASE tool can be used for the development of transformation models, and on the other hand, developing a new code generator for the resulting metamodel [Sch04]. The latter was handled in such a way that the part which depends on the target platform can easily be replaced.

This paper describes this work, and is organized as follows: First, we provide the required background information by summarizing the related MDA standards. Next, the architecture of the code generator is described, which is then illustrated by an example of a model transformation. Finally, conclusions are drawn and potential future work is discussed.

2 MDA Standards

As explained above, in spite of Fujaba's value in validating numerous model management techniques [NZ99,NSW+02,WGN03], the tool lacks standardization. More precisely, its code generator reads its input in a proprietary way, from a non-standard repository, and generates output code for the same repository, again making use of its non-standard API. In what follows, some standards and concepts, which have been used to solve this problem, are presented.

2.1 Meta Object Facility (MOF)

The MOF standard [Obj02] essentially defines a four-layered *metadata architecture*, as shown in Fig. 1. At the top (M3) is the meta-metamodel (also known as the MOF model), a universal language to define metamodels (M2). Metamodels are themselves languages used to define models (M1), which in turn describe the actual data (M0). In other words, the model at level M_n provides a description of some common characteristics of the data at level M_{n-1}. One specific set of data, conforming to a model, is called an *instance* of that model. As shown in Fig. 2, "model" and "metamodel" are relative concepts: a metamodel can easily be seen as a model of a model, while the meta-metamodel can be seen as the model of a metamodel. The Unified Modeling Language (UML) for example, can be formalized by a metamodel (an instance of the MOF model). UML can be used to specify class diagrams, activity diagrams, etc. which, as a consequence, can be parsed to instances of the UML metamodel, or models at layer M1.

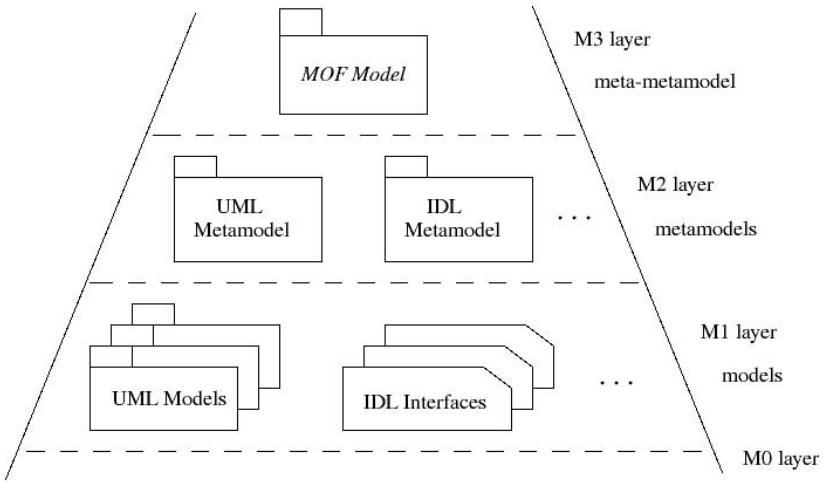


Fig. 1. MOF Metadata Architecture

The MOF model is designed to be universal: it should be adequate to describe *any* metamodel, including its own metamodel (which is the MOF model itself). Since "metamodel" is a relative concept, the meta-metamodel (i.e., the MOF model) can be seen as the metamodel of all metamodels on the M2 layer. In this sense, the MOF model can be stored on the M2 level. However, one can only reason about all metamodels in a standard way by agreeing on one model for meta-metamodeling. Therefore, the MOF model is logically considered to be on the M3 level. In the context of this paper, the most important merit of the MOF standard is that it allows the creation of tools for model analysis and manipulation, which only depend on the MOF model, but not on any specific metamodel. In particular, a MOF repository can be developed, which supports the storage of any MOF-compliant models and metamodels. The open source NetBeans Metadata Repository (MDR [Mic02b]) does just that, and was therefore a logical candidate for the new code generator.

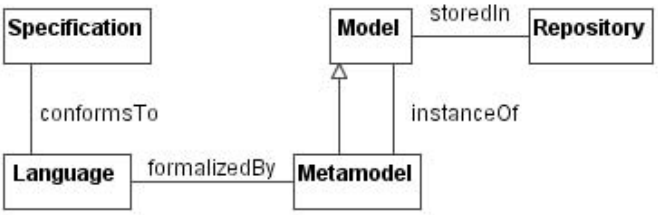


Fig. 2. The relationship between languages, models, metamodels and repositories.

2.2 Java Metadata Interface (JMI)

The MOF standard on itself is not the whole story, since it does not define how models can be accessed from source code. Or rather, it does, but only for CORBA IDL, and not for any other language. As its name suggests, the Java Metadata

Interface (JMI) standard [Mic02a] provides a solution here, by actually *mapping* MOF to Java. More specifically, JMI defines one or more Java entities for each MOF construct, thus introducing a standard API for model access. For example, a MOF class is mapped to two Java interfaces: one "factory" (or "class proxy") interface for constructing objects and one "instance" interface for manipulating them. By applying this mapping to a metamodel, which of course consists of these MOF constructs (as it is a MOF instance), a metamodel-specific set of interfaces is obtained, through which any instance of this metamodel can be accessed and manipulated. In case of UML, for example, these interfaces can be used to add a new UML class to a model of a class diagram, or find an existing UML association and delete it. In addition, there is also a unique set of reflective interfaces, which offers the same possibilities, but without having to use metamodel-specific code.

In order to understand that a standard like JMI is sufficient to build model-manipulating tools in a metamodel- (and model-) independent way, the following two points are crucial:

(i) *model* manipulation must always be carried out through *metamodel* interfaces. For example, a UML class diagram can only be seen in terms of UML classes, UML attributes, UML associations, etc. which are all concepts from the UML metamodel, and as such are present in the UML-specific interfaces.

(ii) indirectly making use of metamodel-specific interfaces, does not make a tool metamodel-dependent. In the following section, it will be shown that the code generator can produce metamodel-specific code by relying on the JMI *mapping rules* only. Obviously, it is desirable that the generated code *is* metamodel-specific, as each model transformation is metamodel-specific as well.

3 Architecture

This section describes the overall architecture of JCMTG, that is, the JMI Compliant Model Transformer Generator, a standards-based alternative for Fujaba's proprietary way of handling model transformations.

3.1 UML Profile for SDM

As already indicated in Section 1, the trouble with SDM (as it is used in Fujaba) is that its syntax as well as its semantics are non-standard, even though they both resemble their UML counterpart. The latter is illustrated in Fig. 3, which displays an excerpt of an SDM specification in Fujaba. While elements of both activity diagrams and collaboration diagrams can easily be recognized, it is clear that no CASE tool is capable of drawing similar diagrams, as UML does not support nesting in that way. Furthermore, the storage of SDM instances in Fujaba is also non-standard. Both aspects of this problem were tackled in JCMTG by designing a UML profile for SDM. In practice, this comes down to *mapping* each SDM construct to a UML alternative. Additionally, stereotypes have been used to dif-

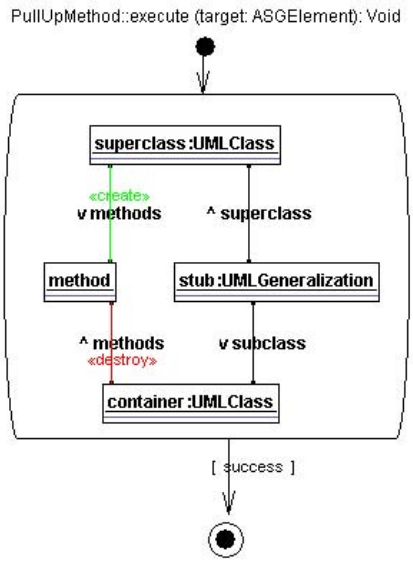


Fig. 3. Example SDM specification edited with the Fujaba UML tool.

ferentiate between several variants of the same basic SDM constructs (for example *forEach* activities versus *code* activities versus normal *story* activities). For the control flow part, this proved to be quite straightforward, because of the support of activity diagrams in UML. For the so-called *transformation primitives*, which actually resemble collaboration diagrams, UML class diagrams have been chosen instead, as these often seem to offer more visual features, such as attribute assignments. An excerpt of the SDM-to-UML mapping is given in Table 1. It should now be clear that the UML profile allows that, on the one hand, SDM specifications can be drawn in any CASE tool, while on the other hand, since UML is a MOF instance, a standards-based MOF repository (in particular, MDR) can be employed for storage purposes.

SDM Construct	**UML Construct**
Story Activity	ActionState
ForEach Activity	ActionState with «for each» stereotype
Unbound object	UmlClass
Bound Object	UmlClass with «bound» stereotype

Table 1
Extract from SDM-to-UML mapping

3.2 Generation of Transformation Code

An overview of the actual code generation process is displayed in Fig. 4. The MOF repository (MDR) plays an important part, and could be seen as the starting point,

as it holds the transformation specification (or transformation model). Since MDR provides a JMI API, this specification can be analyzed in a standardized way by the code generator engine. The open source AndroMDA [Boh03] code generator was chosen for this task, at the heart of which is in fact a set of dynamic content templates. These provide a "skeleton" of the generated code, which is filled in depending on the information in the transformation model.

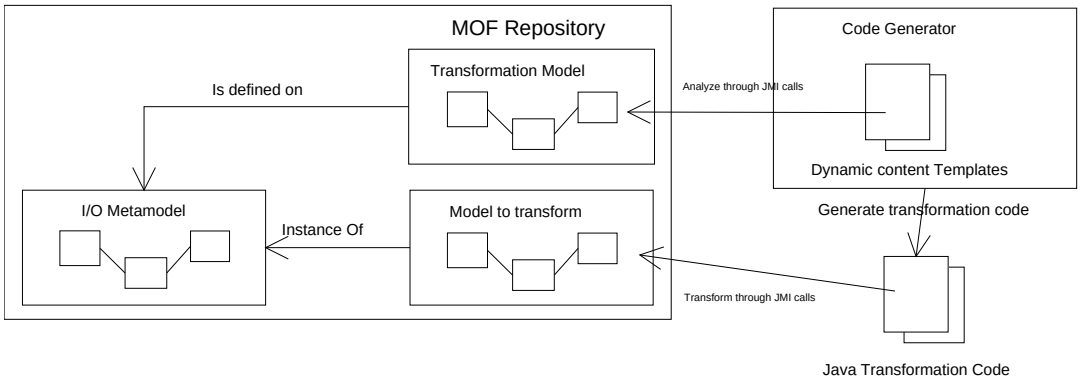


Fig. 4. JCMTG Architecture

Fig. 4 illustrates that this transformation model is defined on the metamodel of the models to be transformed: since we are implementing translations, the in- and output metamodel of the transformations is the same. The artifact resulting from template instantiation is a Java source file, containing metamodel-specific JMI code. This code can analyze and transform any model instantiating this metamodel. In practice, a transformation writer defines model checks and transformations as path navigations and rewritings over a graph structure of this metamodel. One can define a type graph as a class diagram either manually or reverse engineer it from the target repository sources. Ideally, class diagrams of mainstream metamodels (e.g., UML 1.5, 2.0, ...) would be shared by the transformation community.

Note that, even though JMI sets a standard, it may be useful to generate code for other platforms (after all this is MDA). In that case, the dynamic content templates can easily be replaced by a different set, which target a new platform like repositories conforming to the Eclipse Modeling Framework (EMF [MDG+04]).

3.3 Constraints at Two Levels

There are two levels in the transformation process where constraints should be checked.

In order to guarantee generation of correct code, it is important that a transformation model can be checked for well-formedness. Indeed, UML on itself has quite loose semantics, and the interpretation specific to SDM is obviously not captured at all. Ideally, a MOF repository should allow direct verification of such metamodel well-formedness rules (OCL would make a good candidate here, since it is a MOF instance itself), but unfortunately, this is currently not implemented in the MDR. Therefore, the well-formedness of transformation models can currently only

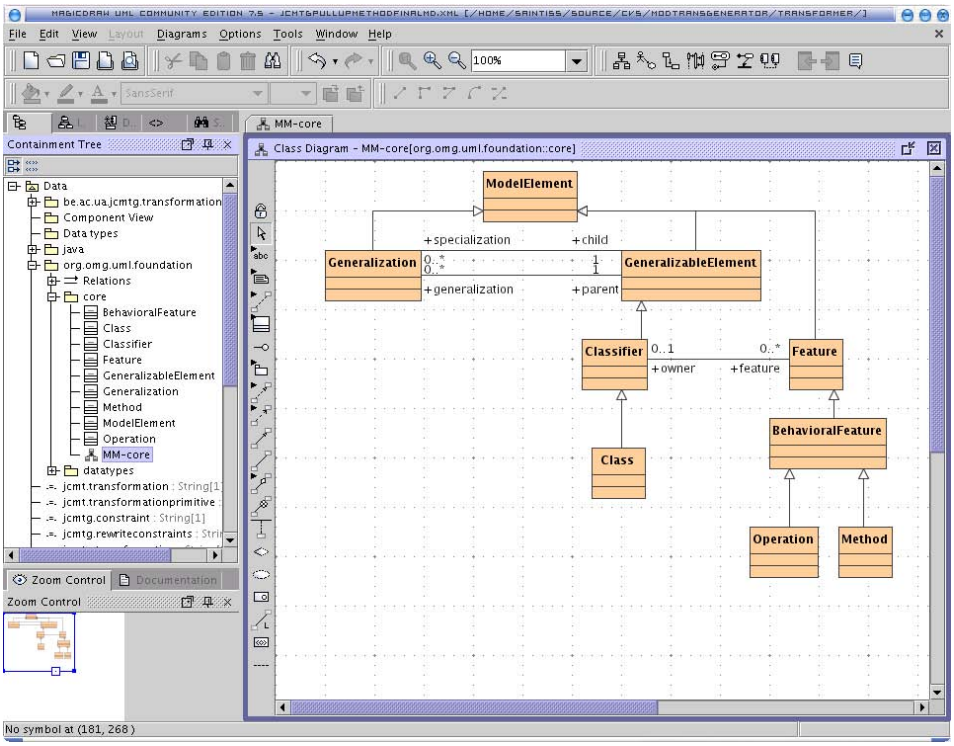


Fig. 5. Fragment of the UML 1.5 metamodel.

be verified after they have been serialized to XMI and imported in a dedicated OCL constraint checker like OCLE [C+04].

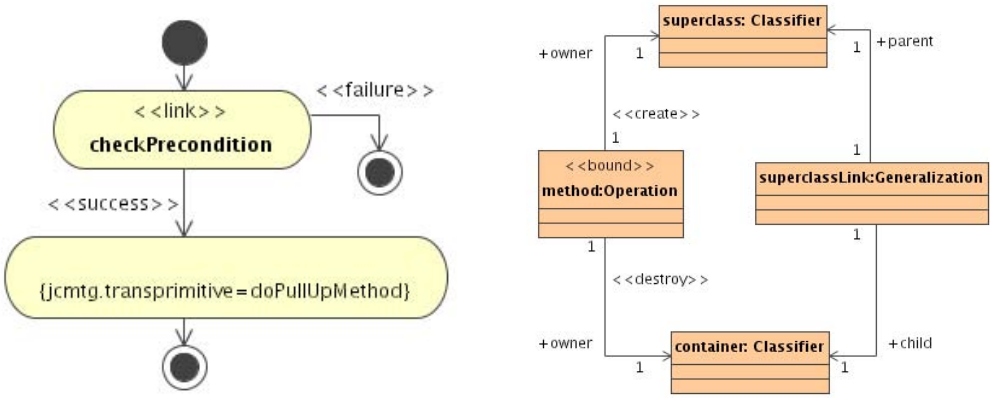
Yet, there is another level where constraints come into play, namely within a transformation specification. Indeed, it may be desirable to only execute (part of) the transformation if a certain complex condition is satisfied, or perform different actions depending on the truth value of such a condition. Thus, OCL is relevant in that context too. However, once more, tool support is lacking. Pragmatically, JCMTG adopted Java conditions instead. Note that this is not ideal, as it makes the transformation model depend upon the target platform (JMI, EMF, ...), which prevents large scale reuse of transformation specifications. The Dresden OCL Toolkit [LO03] seems to be a promising alternative, as it should be capable of parsing OCL constraints, and evaluating them directly on a MOF repository. Unfortunately, the parser is still under development.

4 Example: Pull Up Method

Demonstrating how everything fits together is perhaps best done by means of an example. Consider the so-called “pull up method” refactoring, which basically just moves a method of class A to class B, where A inherits from B. The transformation is defined on a fragment of the UML 1.5 metamodel shown in Fig. 5.

The corresponding transformation model is illustrated in Fig. 6 where, just as in the Fujaba example (Fig. 3 on page 5), two main parts can be distinguished, albeit

not nested anymore. Note that in the UML profile, the reference between the two parts is maintained by means of a tagged value.



(a) (b)

Fig. 6. PUM transformation model

The transformation flow is quite straightforward. First a precondition, which basically just makes sure it makes sense to apply the refactoring, is checked, and only if it returned “true”, the actual transformation is carried out. The latter, the so-called “transformation primitive” specifies a graph rewriting that is displayed in Fig. 6 (b). It illustrates the main idea behind SDM: initially, a set of objects matching the structure given in the primitive, is searched for. More precisely, a method should be found, which belongs to a certain class “container” (this can be checked via the “owner” association in the UML metamodel). Additionally, “container” should have a superclass “superclass”, which can be reached by navigating through the UML metamodel some more. If, and only if, such a structure can be matched, the “owner” link to “container” is removed, and an “owner” link to “superclass” is established, signaling successful completion of the transformation.

After specifying this transformation in a UML 1.5 compliant tool, one exports it to XMI. JCMTG then generates a complete plugin for the Poseidon tool, which has a JMI compliant UML 1.5 repository. Fig. 7 displays the plugin popup appearing when one right-clicks on a method. As specified in the abstract transformation, the method will only be pulled up if the precondition of the refactoring is met.

5 Conclusion and Future Work

It should be clear that the elaborated approach solves the two significant issues from which Fujaba suffers. First, the UML profile ensures the possible usage of any UML 1.5 compliant CASE tool to draw transformation models, as well as standardized model access and storage. Second, the employment of pluggable dynamic content templates guarantees independence from any specific target platform. Nevertheless, JCMTG is only very young, and many aspects would benefit from certain improvements. As already mentioned in Section 3.3 for example, better OCL tool

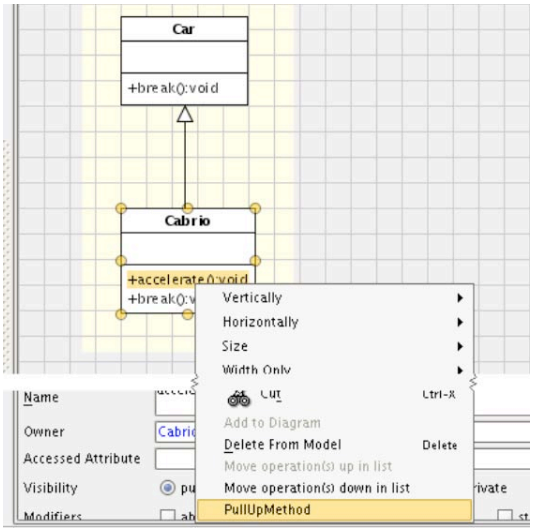


Fig. 7. Screenshot from the UML CASE tool plugin generated from the abstract transformation specification.

support both at metamodel- and model-level, would enable integration of constraint checking. Additionally, expressiveness of SDM could be questioned, especially in the context of inter-metamodel transformations, that is, transforming instances from one metamodel to become instances of another metamodel. In this light, but also when considering very complex transformations, it might be desirable to add additional constructs to the language. Another important issue has to do with the dynamic content templates. Although they can easily be replaced, chances are that a significant amount of any other set of templates would be very similar, if not identical. Therefore, it would probably be worthwhile to investigate how extra levels of abstraction can be introduced between the transformation model and the transformation code. In this light, the transformation engine project at INRIA [dReIeeAI04] is very promising, as it introduces a so-called “pivot-metamodel”. This is a rather low-level metamodel, from which code generation is straightforward. The idea is that a transformation model is first translated to this pivot, instead of generating code immediately. This would ensure that a change of target platform only causes the (trivial) step from pivot to code to be replaced. Finally, a note on the concrete syntax. The argument that the Fujaba notation was more elegant than the, admittedly somewhat artificial, UML notation is probably valid. It is, however, important to distinguish between concrete and abstract syntax. Only the latter is really tied to UML, so nothing prevents a tool developer from creating an environment with Fujaba’s concrete syntax, and transform this behind the scenes to fit into the UML profile for storage. That way, the possibility that for instance an abundance of stereotypes would make transformation specifications less readable, would not be an issue anymore.

References

[Boh03] M. Bohlen. AndroMDA - from UML to Deployable Components, version 2.1.2, 2003.
<http://andromda.sourceforge.net>.

[C+04] D. Chiorean et al. Object constraint language environment (OCLE), version 2.02, 2004.
<http://lci.cs.ubbcluj.ro/ocle>.

[dReIeeAI04] Institut National de Récherche en Informatique et en Automatique (INRIA). MTL Model Transformation Engine, 2004.
<http://modelware.inria.fr>.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)*, volume 1764 of *LNCS*, pages 296–309. Springer Verlag, November 1998.

[GEJ03] Pieter Van Gorp, Niels Van Eetvelde, and Dirk Janssens. Implementing Refactorings as Graph Rewrite Rules on a Platform Independent Metamodel. In *Proceedings of the 1st International Fujaba Days*, University of Kassel, Germany, October 2003.

[Gor04] P. Van Gorp. Write Once, Deploy N: a Performance Oriented MDA Case Study. In *Dagstuhl Seminar 04101 - Language Engineering for Model-Driven Software Development*, march 2004.

[LO03] S. Loecher and S. Ocke. A Metamodel-Based OCL-Compiler for UML and MOF. In *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA*, October 2003.

[MDG+04] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks. International Business Machines, January 2004.

[Mic02a] Sun Microsystems. Java Metadata Interface Specification, June 2002. document ID JSR-40.

[Mic02b] Sun Microsystems. NetBeans Metadata Repository, 2002.
<http://mdr.netbeans.org/>.

[NSW+02] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the* 24^{th} *International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348. ACM Press, May 2002.

[NZ99] J. Niere and A. Zündorf. Using fujaba for the development of production control systems. In *Proc. of International Workshop and Symposium on Applications Of Graph Transformations With Industrial Relevance (AGTIVE), Kerkrade, The Netherlands*, LNCS 1779. Springer Verlag, 1999.

[Obj01] Object Management Group. Model Driven Architecture (MDA), July 2001. document ID ormsc/01-07-01.

[Obj02] Object Management Group. Meta-Object Facility Specification, April 2002. version 1.4. document ID formal/02-04-03.

[Sch04] Hans Schippers. JMI Conforme Modeltransformator Generator. Master's thesis, University of Antwerp, Belgium, 2004.

[SK03] S. Sendall and W. Kozaczynski. Model Transformation - The Heart and Soul of Model-Driven Software Development. *IEEE Software, Special Issue on Model Driven Software Development*, pages 42–45, Sept/Oct 2003.

[WGN03] Robert Wagner, Holger Giese, and Ulrich Nickel. A plug-in for flexible and incremental consistency management. In *Proc. of the International Conference on the Unified Modeling Language 2003 (Workshop 7: Consistency Problems in UML-based Software Development), San Francisco, USA*, October 2003.

The Side Transformation Pattern - making transforms modular and re-usable

Edward D. Willink [1] Philip J. Harris [2]

Thales Research and Technology (UK) Ltd
Reading, England

Abstract

The basic principles of meta-modelling are now well established for individual models. Activities such as the MOF QVT [5] are now extending these principles to transformation between models. However, meta-model incompatibilities between transformations reduce opportunities for effective re-use, hindering wide scale adoption. We introduce a pattern, the *Side Transformation Pattern*, that arises naturally as transformations are made re-usable, and present a series of examples that show how its use can bring clarity and robustness to complex transformation problems.

Key words:
MDA,MOF,Modularity,Pattern,QVT,Transformation,UMLX.

1 Introduction

Transformation is often presented as a complete activity. This impression is particularly prevalent in today's 'MDA-compliant' tools, which perform a one-stage rewrite of source (Platform Independent Model) concepts into target (Platform Specific Model) concepts. Such an approach has no explicit Platform Model as required by the Model Driven Architecture [4], and offers little opportunity for evolution through progressive transformation.

Where a single stage approach satisfies the user's requirements, it can offer dramatic improvements in productivity, for example in code generator templates that produce Java or C++ source text directly from UML graphical models. Too often however, the one-stage approach inhibits a proper separation of different programming concerns. The result is a lack of flexibility and visibility in the transformation process, leading to poor opportunities for enhancement or re-use.

[1] Email:EdWillink@iee.org
[2] Email:Phil.Harris@thalesgroup.com

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Once transformation is viewed as a multi-stage programming discipline, traditional software engineering issues such as modularity and re-use arise. In the context of transformations, re-use is hampered by incompatibilities between the meta-models in use.

If a transformation-driven approach with the richness required by many different user applications is to be widely accepted, both engineering issues and programming concerns must be addressed in the translation of increasingly abstract source representations to a variety of different platforms.

We are interested in transforming Domain Specific Visual Languages (DSVL) that specify required computations and communications into efficient executable code. The executable code may be either C or VHDL, depending on whether the target platform is sequential or concurrent in nature. Many distinct transformations are required - some that resolve graphical abstractions and instantiation hierarchy can be shared by both targets, others, such as establishing a sequential schedule or synchronising concurrent execution, are very target dependent. The complete set of transformations is much too complicated to express as a single pass, and many, such as state machine synthesis, overlap with more conventional functionality.

We therefore look to an approach that involves many small transformations, each dealing with a single aspect of the problem, such that relevant transformations extracted from a large library of re-usable transformations can be exploited to build an apparently custom composite with few truly custom contributions. Such an approach also offers great advantages in the validation of the complete transformation, control of the transformation code base, and the transfer of system knowledge.

Even within the UML to Java world, we see benefits in using this approach to progressively realise the more abstract UML constructs into simple Object Oriented concepts, which are then reified within the limitations of the Java meta-model before the final conversion to text. Such a progression can readily evolve to accommodate additional Domain Specific Visual concepts at the start, alternate Object Oriented perspectives in the middle, or specialised code metrics at the end.

In this paper, we first present a transformation pattern that recurs as we seek to make our transformations modular and re-usable. These are essential features for a successful MDA [3]. We then outline three different examples of the pattern in our compilation system. The third example comprises three distinct invocations of the pattern demonstrating how the pattern may be composed sequentially, concurrently and recursively.

If a transformation is to be re-usable there are three possibilities

- standardise the meta-model on which the transformation operates
- translate the transformation to the custom meta-model
- translate to and from transformation-specific meta-models.

Standardising the meta-model is not practical, since it is unlikely that any

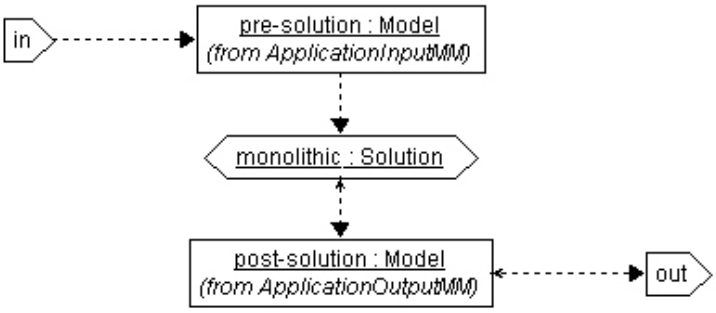


Fig. 1. A Monolithic Transformation.

future version of UML will be sufficiently lightweight for typical usage yet flexible enough for all.

Translating the transformation may be the best option. But we must wait until tools evolve to support transformations on transformations, and until our transformations for re-use are written in transformable languages. QVT looks very promising here.

Projecting the relevant elements of the custom meta-model into the transformation domain is practical now, and may allow optimisation in the future. This approach forms the basis of the *Side Transformation Pattern*; a pattern that, analogously to a side calculation, performs some subsidiary action in order to make overall progress.

We use a pattern form [2] in order to provide a clear concise exposition of the problem, context, forces and solution. As such, the aim is to capture expertise, rather than to claim any great novelty. Many readers will surely recognise the approach, and, like the authors, may regret that they had not always had the discipline or tools to apply the pattern more often.

2 The Side Transformation Pattern

Context

You are designing or programming an application. You encounter the need for a transformation that solves a recurring or complex problem. You find that re-use requires variation in the different input or output meta-models.

The monolithic solution to this problem demonstrates this context and a way of extending UML to support describe transformations.

In Figure 1 (based on the proposed UMLX notation [6]) rectangles use UML class instance notation to show models and their meta-models, lozenges denote transformation instances, and pointed rectangles identify the input and output ports of the overall transformation component. The arrows show the 'data' flow of models between ports, models and transformations.

Note that although both input and output are shown as instances of `Model`,

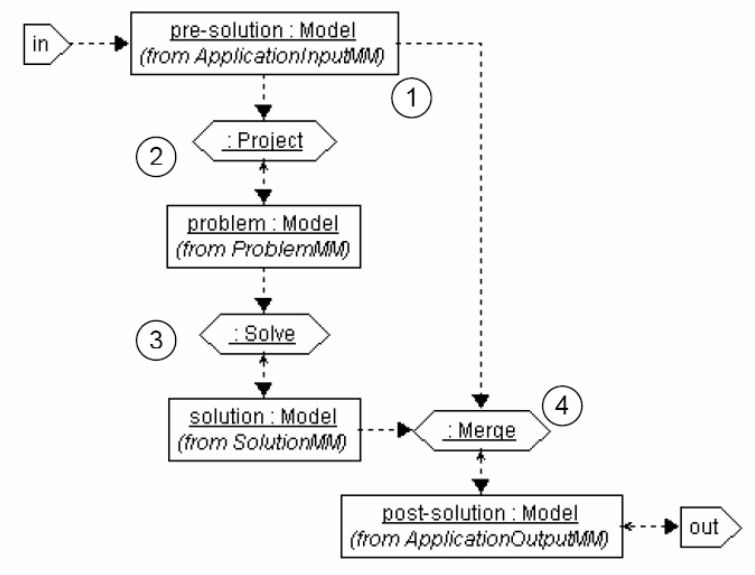


Fig. 2. Side Transformation Pattern.

each comes from a different package, and so may be radically different.

Problem

How do you make your transform re-usable and modular?

Forces

Monolithic implementation of the transformation avoids complexity.

Monolithic implementation couples the solution to the business rules and meta-models. This may create problems by:

- inhibiting re-use in other applications
- inhibiting re-use with other meta-models
- inhibiting re-use with other business-rules

A solution to the recurring problem may be available from another programming domain.

The transformation may build on existing transformations.

Solution

Isolate the solution to the recurring problem from the surrounding application in a side transformation (Figure 2).

The solution involves four activities:

1 *Fork.* The incoming instance of `Model` from the `ApplicationInputMM` (meta-model) is made available to two separate transformation paths, one for the side transformation and another for the eventual merge.

2 *Project*. A view of the `Model` instance from `ApplicationInputMM` is created that extracts the salient information from the application and expresses it as an instance of `Model` from the solver's `ProblemMM`.

3 *Solve*. The re-usable solution is applied to transform the problem expressed as an instance of `Model` from the solver's `ProblemMM` into an instance of `Model` from the solver's `SolutionMM`.

4 *Merge*. The second copy of the incoming instance of `Model` from `ApplicationInputMM` is updated or rewritten to incorporate the results of the side transformation resulting in an instance of `Model` from the `ApplicationOutputMM`.

(Note that the instance of `Model` from each meta-model represents the root of the meta-model instantiation, and that each meta-model may be completely independent. There may therefore be four different `Model` classes.)

Resulting Context

Separate meta-models are associated with the input and output of the re-usable solver. The solver is therefore independent of the application.

The *Project* transformation is responsible for creating the `problem` view. The stimuli from the business rules are therefore isolated in a separate module.

The *Solve* transformation is independent and re-usable.

The *Merge* transformation is responsible for interpreting the `solution`. The responses to the business rules are again isolated in a separate module.

Variations

Meta-models may be re-used; each pair of `ProblemMM`, `SolutionMM` and `ApplicationInputMM`, `ApplicationOutputMM` meta-models may be the same or derived from a shared core meta-model.

The pattern may be applied recursively, with *Project*, *Solve* and *Merge* as further instances of the pattern.

The pattern may be applied concurrently with multiple *Solve*s contributing to a more complicated *Merge*.

3 Example 1: Type Inference Constraints

In our first example we consider a compilation stage for a DSVL that defines a system as a connection of components. The important parts of the model are

- actors - components that encapsulate their scheduling
- ports - the communication interfaces of actors
- connections - the communication paths between ports

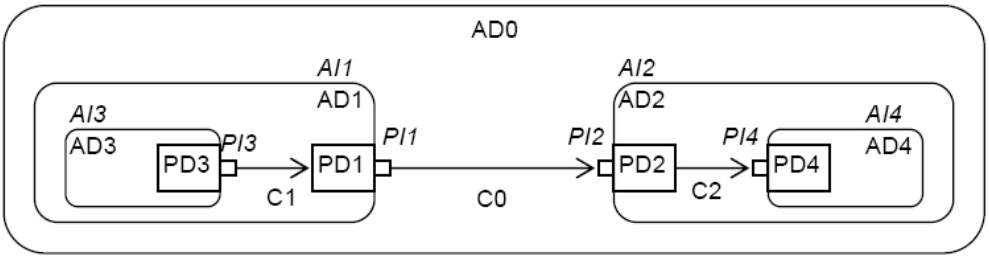


Fig. 3. Conflated actor hierarchy.

The model is hierarchical, so at some hierarchical level within an actor definition (AD), we may have connections between the port definitions (PD) of the defined actor and port instances (PI) of the actor instances (AI) instantiated within the defined actor.

In order to make this example accessible to a wider audience, we use UML 2.0 activity diagram notation rather than a DSVL for Figure 3. Actor instances are therefore shown as activities (rounded rectangles), whose external port (instances) are shown as pins (small rectangles) and internal port (definitions) are shown as parameters (large rectangles).

Figure 3 conflates two levels of structural hierarchy to highlight the hierarchical nature of instantiation. The outer actor definition $AD0$, comprises instances $AI1$ and $AI2$ of $AD1$ and $AD2$, each of which has a single port $PD1$ and $PD2$ respectively instantiated as $PI1$ and $PI2$. $AD1$ and $AD2$ in turn instantiate $AD3$ and $AD4$. A single connection traverses the hierarchy from $PI3$ of $PD3$, via $C1$ to $PI1$ of $PD1$, via $C0$ to $PI2$ of $PD2$, and via $C2$ to $PI4$ of $PD4$.

In order to generate code for such a system, it is important to know properties such as the type of each port and each connection. It is clearly undesirable for the programmer to have to define the type at each of the 11 nodes, so we can exploit the constraint that the type must be consistent at all nodes to infer missing declarations, and to validate redundant declarations.

Within each level of hierarchy we have a constraint for the connections:

$$T_{PI1} = T_{C0} = T_{PI2}, T_{PI3} = T_{C1} = T_{PD1}, T_{PD2} = T_{C2} = T_{PI4}$$

(where T_{C0} denotes the type of connection $C0$), and between hierarchical levels we have:

$$T_{PD1} = T_{PI1}, T_{PD2} = T_{PI2}, T_{PD3} = T_{PI3}, T_{PD4} = T_{PI4}$$

and at some point we may have e.g.

$$T_{PD3} = INTEGER$$

These constraints may be readily expressed as instances of the meta-model shown in Figure 4. The `Model` comprises `Constraints` that require each of their `Terms` to be compatible. A `Literal` enforces a specific value such as $INTEGER$. An `Equality` associates a free variable such as T_{C0}. The

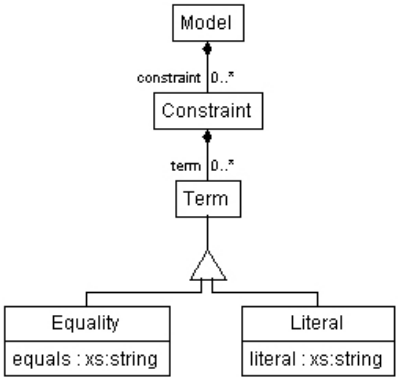


Fig. 4. Constraint solution meta-model.

problem therefore involves many constraints that (hopefully) intersect at a `term.equals`.

The result is an instance of the input meta-model that satisfies the input/output requirement that each distinct incoming term element appears in the output exactly once, and that each pair of term elements that share a common parent in the input also share a parent in the output.

Practical implementation of these type inference algorithms requires awkward recursive traversal up and down the design hierarchy. A monolithic implementation therefore results in a complicated solution that is very tightly coupled to the design meta-model, with little opportunity for re-use. However, the *Side Transformation Pattern* allows encapsulation of each of these separate concerns, easing implementation and maintenance:

- The `ApplicationInputMM` is as required.
- The *Project* transformation creates the system of type (in)equalities from the input meta-model.
- The `ProblemMM` is as shown in Figure 4.
- The *Solve* reduces the system of (in)equalities into a list of unique types.
- The `SolutionMM` is as shown in Figure 4, with an additional prohibition on multiple `Term` parents.
- The *Merge* annotates each node in the original design with its inferred type.
- The `ApplicationOutputMM` is the same as the input, except that all optional type related attributes have been resolved.

The type solver is now decoupled, and combines elements together without concern for the hierarchical complexities of the DSL or the semantic validity of the solution. The semantic interpretation of an unresolved constraint (a `constraint` without a `term.literal`) or a conflicting constraint (a `constraint` with more than one distinct `term.literal`) is resolved as part of the merge of the solution back into the original model.

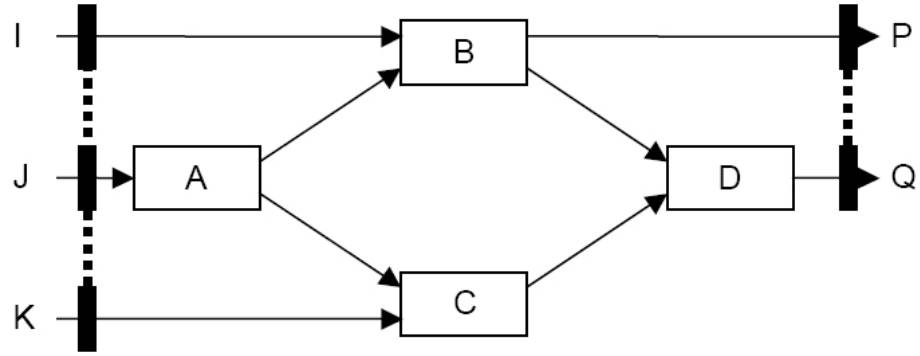


Fig. 5. Example flow graph.

4 Example 2: Common Feature Elimination

Common Sub-expression Elimination is a standard compiler optimisation that seeks to eliminate repeated computations by caching and re-using the result of a single computation. The optimisation can be applied more generally in a modelling context so we refer to feature rather than sub-expression elimination.

A naive implementation of such an optimisation could incur quadratic complexity over the full model as an outer search identifies each candidate for elimination, and as an inner search identifies duplicates.

A more efficient implementation uses the *Side Transformation Pattern*.

- A linear complexity *Project* transformation builds an instance of a `ProblemMM`, which identifies all the candidates.
- The *Solve* transformation builds a smaller instance of the `SolutionMM` identifying the common features.
- A linear complexity *Merge* transformation rewrites the input to exploit the common features.

Separation of the three *Project*, *Solve* and *Merge* transformations isolates the three distinct activities, provides an opportunity to share the *Solve* transformation, and results in near linear complexity (if a suitable hashing algorithm is used by the *Solve* transformation).

5 Example 3: Synchronisation Barrier Insertion

Our third example concerns automatic placement of synchronisation barriers within a repetitive computation defined by a flow graph. Figure 5 provides an example, with rectangles encapsulating a nested (hierarchical) computation, thick bars denoting synchronisation barriers and directed arrows showing the data flow.

(This graph demonstrates a DSVL. An equivalent UML activity diagram

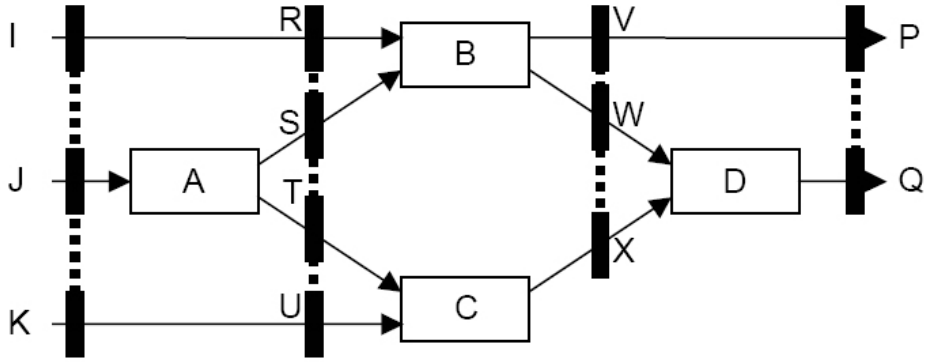


Fig. 6. Example pipelined flow graph.

would require each actor rectangle to be elaborated by an activity with pins such that all inputs are synchronised, and all outputs synchronised. The synchronisation barrier must similarly be elaborated by a null activity with synchronised inputs, and synchronised outputs.)

Each synchronisation barrier captures its input at the end of one computation cycle and propagates it at the start of the next. The barriers therefore ensure that the $\{I,J,K\}$ inputs and $\{P,Q\}$ outputs constitute coherent tuples.

During one complete evaluation the inputs I, J, K and the current states of the sub-processes A, B, C, D are used to compute the outputs P and Q and the new states of A, B, C, D. The precedence constraints (data dependencies) in the flow graph require A to compute before B and C (which may compute in parallel). B and C in turn compute before D completes the overall computation.

The peak computation rate is determined by the slowest path: a directed route through the computation graph starting at, and ending at, a synchronisation barrier. There are five such paths in Figure 5, of which clearly one of the two paths from J to Q is slowest, so the computation time in the example is $t_{max} = t_A + max(t_B + t_C) + t_D$ where t_D is the computation time of node D, and t_{max} is the worst case computation time. In a naive implementation, this computation time would limit the maximum throughput to $1/t_{max}$ computations per second.

However, since our target platform provides separate resources (parallelism) for each of A, B, C and D, a higher throughput can be achieved by introducing additional barriers so that $\{A\}$, $\{B,C\}$ and $\{D\}$ are computed during successive execution cycles. Successful implementation of this requires the introduction of the seven further barriers (R, S, T, U, V, W and X) as shown in Figure 6.

The number of barriers to insert on each data flow is often obvious for simple designs such as Figure 5. However, the problem becomes more difficult when complex dependencies arise, such as a need to insert further barriers

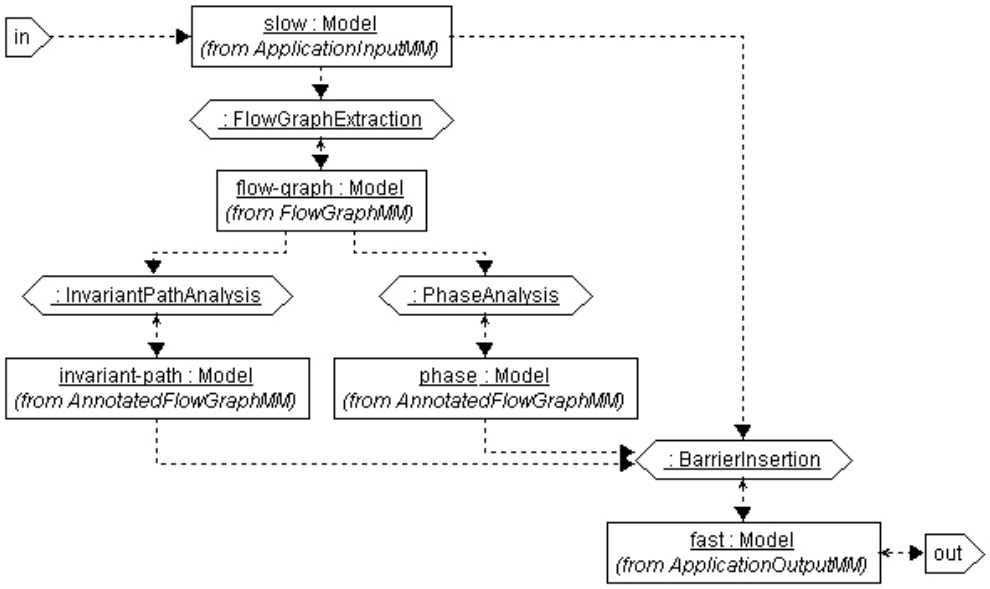


Fig. 7. Synchronisation barrier insertion.

within C. A fully automated solution is very desirable.

A valid solution satisfies the constraints:

- each path satisfies the specified throughput
- coherent inputs and outputs remain coherent

An 'optimum' solution may require compromises between:

- the costs of inserted barriers
- design margin for the specified throughput
- minimum delay at the outputs

It is not appropriate to elaborate the algorithm details here. Suffice it to say that the solution can be captured by a set of integers that specify how many cycles late each actor computes, and that the problem can be represented by a set of inequalities that must be satisfied by these integers.

The solution is established recursively, so that the synchronisation barriers required by inner actors are resolved and accommodated by the solution for outer actors. The solution for a single recursion uses three instances of the *Side Transformation Pattern* of Figure 2.

The top level, shown in Figure 7, projects out a meta-model instance that focuses on the flow-graph details. This is then fed to two different solvers. One identifies constant paths for which synchronisation barriers are redundant, and the other identifies the necessary delays on each non-constant path. The results of these two analyses control the subsequent rewrite of the input model incorporating the required synchronisation barriers.

Each of the solvers is also an instance of the *Side Transformation Pattern*.

The invariant path analysis is comparatively simple and so uses a degenerate form of the pattern with the same meta-model throughout and consequently a null *Project* transform. The *Merge* merely adds an *is-constant* attribute to selected flow-graph nodes.

The phase analysis is more sophisticated and fully exploits the *Side Transformation Pattern* in a similar way to the first example. The *Project* transform creates a system of inequalities describing the system constraints for the solver to act on. The *Solve* is based on PIP [1] the *Parametric Integer Programming* solver. This generates its own output meta-model describing the solution. The results are again merged by annotating the flow-graph model.

The final barrier insertion *Merge* now has two models to combine with the design model from the original *Fork*. The *Merge* is a design rewriter that inserts synchronisation barriers into non-constant paths in the computation graph. The input and output meta-models are identical, however the output has been optimally pipelined on a global basis to satisfy the required computation throughput.

Once again the pattern enables independent solvers to be used, in this case an in-house invariant path identification algorithm, but more importantly a highly optimised generic solver with proprietary input and output meta-models (C structures) from a third-party. The use of a high-level *Side Transformation* to encapsulate the two solvers and merge their results isolates each of; the mechanism for solving the problem, the description of the design that contains the problem, and the final application of the solution to that design.

6 Acknowledgements

The authors are grateful to Thales Research and Technology for permission to publish this paper, much of which is based on work done on the SPEAR and GSVF-2 projects.

7 Conclusion

We have shown how a particular pattern of fork, projection, computation and merge enables a collection of re-usable transformations to be effectively combined into a larger composite transformation. This *Side Transformation Pattern* promotes encapsulation and isolation of a solver from the application meta-model and its attendant business rules.

The introduction of this pattern and its degenerate forms can also assist in the formal description of transformation processes by clearly defining boundaries where re-use, validation, and optimisation are possible.

As examples, we have described parts of a Model Driven compilation system that converts from a visual representation of a system to executable code for a variety of platforms by the use of many small progressive transformations

based on the pattern. In doing so, we have demonstrated that the pattern is scalable and self-consistent, and that the four major operational phases identified in this pattern concisely describe fundamental transformation activities, aiding general discussion of transformation behaviour.

References

[1] Feautrier, P., J. F. Collard, and C. Bastoul, "Solving systems of affine (in)equalities", Technical Report, PRiSM, Versailles University, 2002, URL: http://www.prism.uvsq.fr/~cedb/bastools/download/abe01_MHAOTEU.ps

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

[3] Lemesle, R., "Meta-modeling and modularity : Comparison between MOF, CDIF & sNets formalisms", URL: http://www.sciences.univ-nantes.fr/info/lrsg/Recherche/mda/Lemesle.CDIF.pdf

[4] Miller, J., and J. Mukerji, ,"MDA Guide Version 1.0.1", OMG Document omg/2003-06-01, URL: http://www.omg.org/docs/omg/03-06-01.pdf

[5] QVT-Merge Group, "Revised submission for MOF 2.0 Query/Views/Transformations RFP (ad/2002-04-10)", OMG Document ad/04-04-01, URL: http://www.omg.org/docs/ad/04-04-01.pdf

[6] Willink, E. D., *A concrete UML-based graphical transformation syntax - The UML to RDBMS example in UMLX*, Workshop on Metamodelling for MDA, University of York, England, 24-25 November 2003, URL: http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/doc/umlx/M4M03.pdf

An Algebraic Baseline for Automatic Transformations in MDA

Artur Boronat [1] José Á. Carsí [2] Isidro Ramos [3]

Information Systems and Computation Department
Polytechnic University of Valencia
Valencia, Spain

Abstract

Software evolution can be supported at two levels: models and programs. The model-based software development approach allows the application of a more abstract process of software evolution, in accordance with the OMG's MDA initiative. We describe a framework for model management, called MOMENT, that supports automatic formal model transformations in MDA. Our model transformation approach is based on the algebraic specification of models and benefits from mature term rewriting system technology to perform model transformation using rewriting logic. In this paper, we present how we apply this formal transformation mechanism between platform-independent models, such as UML models and relational schemas. Our approach enhances the integration between formal environments and industrial technologies such as .NET technology, and exploits the best features of both.

Key words: Graph-based models, MDA and model transformation, consistency and co-evolution, term rewriting systems.

1 Introduction

The development of information systems is getting increasingly complex as these systems become both more widely distributed and pervasive in influence [1]. New technologies that enable these capabilities allow for a wide range of choices which the software developer must take into account. Such choices involve technologies for describing software such as object-oriented programming languages, XML, database definition, query languages, etc. These technologies

[1] Email: aboronat@dsic.upv.es
[2] Email: pcarsi@dsic.upv.es
[3] Email: iramos@dsic.upv.es

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

have different levels of abstraction. For instance, there are technologies that support requirements engineering such as DOORS or RequisitePro. There are also technologies that support modeling approaches such as UML and others that permit the implementation of a specific solution such as .NET Framework, Java, etc. On the one hand, modeling approaches provide mechanisms to find the most effective representation of real-world concepts in the domain space of a software project. On the other hand, the majority of technical frameworks offer a large number of mechanisms to build solutions in the computer space.

In general, the initial idea was that the models had to represent reality from the user perspective, starting from the problem space. However, these models actually come from the solution space (such as UML) abstracting the features of object-oriented programming language and constraining the logic representation of the problem space. This approach was taken in order to be able to automatically generate software artifacts from a layer that is more abstract than the final code. Unfortunately, the automatic generation of software has not yet reached maturity, provoking the crisis of the CASE tools that appeared in the mid 90s. Contrary to what we expected ten years ago, we have arrived to a complicated situation where the design and implementation of a software product requires an enormous effort involving several technologies.

However, all the software artifacts that we have mentioned above can act as models with a specific level of abstraction. The generation of code that CASE tools are supposed to perform (model compilers) can also be considered as the transformation of a model of a high level of abstraction into one with a more specific level of abstraction. In accordance with this approach, a research field has emerged, providing a solution to problems of this kind: model management. A model is an abstract representation of a domain that enables communication among heterogeneous stakeholders so that they can understand each other. Model management [2] aims at solving problems that require model representation and its manipulation in an automated way.

The OMG's Model Driven Architecture (MDA) initiative [3] is set in this context and provides several proposals to define models, through the standard Meta-Object Facility (MOF) [4]. It also offers proposals to perform model transformations by means of the Query/View/Transformations (QVT) language, which is still in its early stages [5]. While a lot of attention has been given to the transformation of platform-independent models into platform-specific models, the scope of MDA goes beyond this in an attempt to model all the features of a software product throughout its life cycle. Nonetheless, a precise technique to provide formal support for the entire process of model transformation has not yet been developed.

The MOMENT (MOdel manageMENT) platform follows this trend by providing a framework where models can be represented using an algebraic approach. MOMENT is based on an algebra that is made up of *sorts* and operators that permit the representation of any model as a term that can be automatically manipulated by means of operators. The MOMENT Framework

benefits from the best features of current visual CASE tools and the main advantages of formal environments such as term rewriting systems, combining the best of both industry and research.

This paper presents the generic model transformation mechanism of the MOMENT Framework, and its application to platform-independent models, obtaining a UML model from a relational schema in the MDA context. The paper is structured as follows: Section 2 presents other approaches to provide automatic support for model transformations; Section 3 indicates an example to illustrate the transformation process through the paper; Section 4 provides an overview of the MOMENT Framework; Section 5 presents the model transformation mechanism of the MOMENT Framework, focusing on the use of a Term Rewriting System (TRS) as a formal environment to perform automatic model transformations. Finally, Section 6 summarizes the work and indicates the future directions of our research tasks.

2 State of the Art

The essentials of a model theory for generic schema management are presented in [6]. This model theory is applicable to a variety of data models such as the relational, object-oriented, and XML models, allowing model transformations by means of categorical morphisms. RONDO [2] is a tool based on this approach. It represents models by means of graph theory and a set of high level operators that manipulate such models and the mappings between them by using category theory. Models are translated into graphs by means of specific operators for each metamodel. These algebraic morphisms are implemented using imperative algorithms such as CUPID [7]. CUPID is an algorithm for matching schemas in the RONDO tool.

In the MOMENT platform, we follow the framework that is proposed in the Meta-Object Facility specification (MOF). MOF is one of the OMG family of standards for modeling distributed software architectures and systems. It defines an abstract language and a four-layer framework for specifying, constructing and managing technology neutral metamodels. A metamodel is an abstract language for different kinds of metadata. MOF defines a framework for implementing repositories that hold metadata described by the metamodels. This framework has inspired our platform for model management, although we do not use the same vocabulary to describe metamodels. In the case of MOF, the two most abstract layers offer an abstract view of a specific model. This allows for the definition of generic operators to manipulate models and metamodels.

The MétaGen project [9] has dealt with model engineering since 1991, aiming at a fully automatic generation of a conventional application from a description given by its intended user. Such a description is performed by means of PIR3, a variant of what is known in the Database community as Entity-Relationship Model. In [10], Revault et al. compared three metamod-

eling formalisms to share the experience they acquired during the MétaGen project. In that paper, they presented a way to transform MOF-based metamodels into PIR3-based metamodels so that the metamodels could benefit from the MétaGen tools. This proposal constrains the expressivity of the source metamodels because the Object-Oriented paradigm is richer than the Entity-Relationship paradigm. Our model management approach supports the definition of several metamodeling languages such as MOF or PIR3, considering them as models at the same abstraction level and using generic operators. This makes automated transformations between models of both metamodels possible, without loss of expressivity.

XML [18], the standard for data communication between applications is also used to represent models and metamodels by means of the XMI specification [19]. MOF defines a meta-metamodel, while XMI indicates the physical representation for the metamodels and the models that can be defined with it. In the model transformation field, there is a XML specification that allows the transformation of a XML document into another one, called XSLT [20]. It could be used to transform models that are represented in the XML format. Comparing XSLT to term rewriting systems, there are some differences that should be pointed out:

- Although XSLT is said to be a declarative language, control instructions, such as jumps and loops, can be used to guide the transformation process. In contrast, a term rewriting system takes over the transformation rule evaluation process.
- Writing an XSLT program is a long and painful process which implies poor readability and high maintenance cost for associated programs. Also, writing an XSLT program requires good skills in the MOF and XMI specification, because when using XPATH and XSLT, the developer must take into account the structure of models which depends on metamodels. These metamodels are in turn widely influenced by MOF and XMI. By expressing metamodels as algebras, we can deal with a more specific syntax that reflects their semantics using the algebras as domain specific languages. Therefore, writing models (and consequently transformation rules) becomes easier and more comprehensible.
- Transformation rules in XSLT are applied without taking into account the target XML schema (metamodel when transforming models), implying a posterior checking to determine whether the obtained document is a valid XML document that conforms to the target schema. Rewriting rules in an algebra take into account the source and the target algebras so that a posterior checking is no longer needed.
- Executing an XSLT program is not user-friendly for model transformation because there are no error messages to advise the user about an incorrect transformation.

The MTRANS Framework [21] provides an abstract language to define

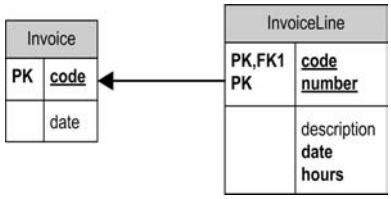


Fig. 1. Relational schema

transformation rules that are compiled to XSLT. Even though this language is more compact and easier to understand than XSLT, it still keeps instructions to manage the transformation rules evaluation. Nevertheless, transformations using XML technology imply the use of standard specifications that are industrially supported while term rewriting systems usually remain within the field of research.

3 A Running Example

Consider a car maintenance company that has worked a long time for a large car dealership. The maintenance company has always worked with an old C application where the information is stored in a simple relational database that does not take into account integrity constraints. The car dealership has recently acquired the car maintenance company and the president has decided to migrate the old application to a new OO technology in order to improve maintenance and efficiency. Therefore, the target application will be developed by means of an OO programming language.

Suppose that a part of the original database are two tables related by means of a foreign key, representing the information of an invoice and its lines, as shown in Fig. 1. To obtain a UML model that is semantically equivalent to this relational schema, a designer usually builds it manually, which involves high development costs, since the entire initial database must be taken into account. What is worse is that this process is error-prone due to the human factor.

4 The MOMENT Framework

The MOMENT (MOment manageMENT) Framework is a modular architecture divided into the three traditional layers: interface, functionality and persistence. In each one of them, the environment benefits from mature tools, such as graphical CASE tools at the interface layer, term rewriting systems at the functionality layer, and RDF repositories at the persistence layer. Hence, the MOMENT Framework aims at using the best features of each environment, bringing industrial modeling tools closer to more formal systems. Fig. 2 shows an overview of the MOMENT Framework.

The functionality layer permits the representation of models and the per-

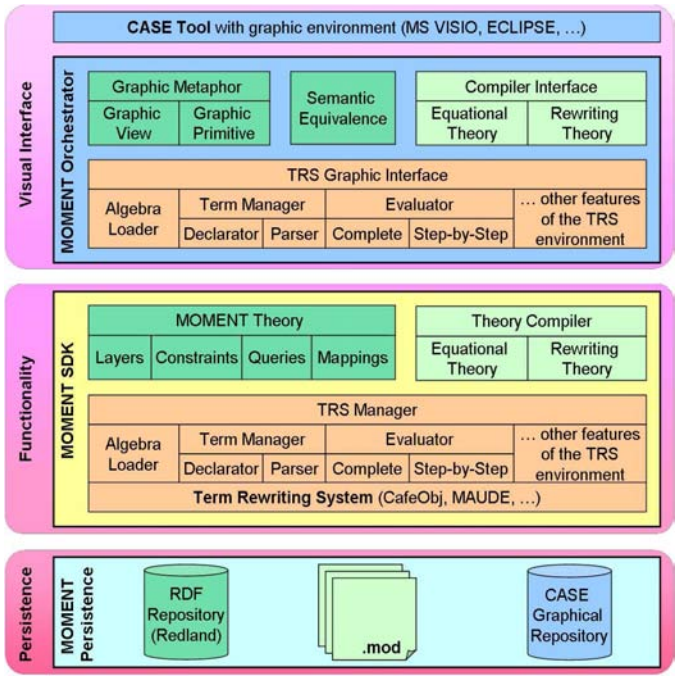


Fig. 2. The MOMENT Framework

formance of transformations over them. The core of the functionality layer is a module called *MOMENT Theory*, which allows model representation and manipulation by means of an algebraic approach. We use the expressiveness of the algebra that the platform is based on to define and represent a model as an algebraic term. This algebra represents models by means of terms of a *sort* called *Schema*. These terms are made up of by concepts and properties. The concepts are the main entities of a model, and the properties either describe them with values or establish relationships between them. The properties contain information about cardinality, indicating how many concepts can be related to the owner of the property.

The MOMENT platform uses several metadata layers to describe any kind of information including new metadata types. This architecture is based on both the classical four-layer metamodeling architecture (following standards such as ISO [11] and CDIF [12]) and on the more modern four-layer framework proposed in the MOF specification [4]. In our work, we divide the platform into four abstract layers:

- The M0-layer collects the examples of all the models, i.e., it holds the information that is described by a data model of the M1-layer.
- The M1-layer contains the metadata that describes data in the M0-layer and aggregates it by means of models. This layer provides services to collect examples of a reality in the lowest layer.
- The M2-layer contains the descriptions (meta-metadata) that define the structure and semantics of the metadata located at the M1-layer. This

layer groups meta-metadata as metamodels. A metamodel is an "abstract language" that describes different kinds of data. The M2-layer provides services to manage models in the next lower layer.

- The M3-layer is the platform core, containing services to specify any metamodel with the same common representation mechanism. It is the most abstract layer in the platform. It contains the description of the structure and the semantics of the meta-metadata, which is located at the M2-layer. This layer provides the "abstract language" to define different kinds of metadata.

The *MOMENT Theory* module also provides a mechanism to define transformations between metamodels. The *TRS Manager* module wraps a TRS, which carries out the model transformation by applying a set of rewriting rules automatically. We have used the CafeOBJ environment as TRS [13]. The *Theory Compiler* module permits the compilation of the algebraic specification of a metamodel into a theory based on equational logic. It also compiles the defined mappings between the elements of the metamodels into a theory based on rewriting logic in order to perform the model transformation on the wrapped TRS.

Some of these modules have been developed using the functional language F# [14], which provides convenient features to work with algebraic specifications and with imperative programming environments such as .NET technology. A combination of functional languages and algebraic specification languages has permitted us to reach our goals. On the one hand, the MOMENT algebra is implemented in F#, which provides efficient structures for navigation and specification manipulation. On the other hand, a TRS provides a suitable environment to support automatic model transformation.

5 PIM-to-PIM Transformation

MDA raises the level of abstraction in the software development process by treating models as primary artifacts. Models are defined using modeling languages, but when those languages are intended to be used for anything more sophisticated than drawing pictures, both their syntax and their semantics must be specified. In this case, the use of formal languages usually involves dealing with their complex syntax, making them unpopular in industry. In this sense, the MOMENT Framework is user-friendly and permits the use of formal techniques from well-known CASE tools to both define models by means of algebraic specifications and to perform model transformations using rewriting logic [15].

In this section, we present how MOMENT provides formal support for generic model transformation in the MDA context, by generating a UML model from a relational schema. First, we explain a general overview of the transformation mechanism, and later, we focus on the most formal phases of the process.

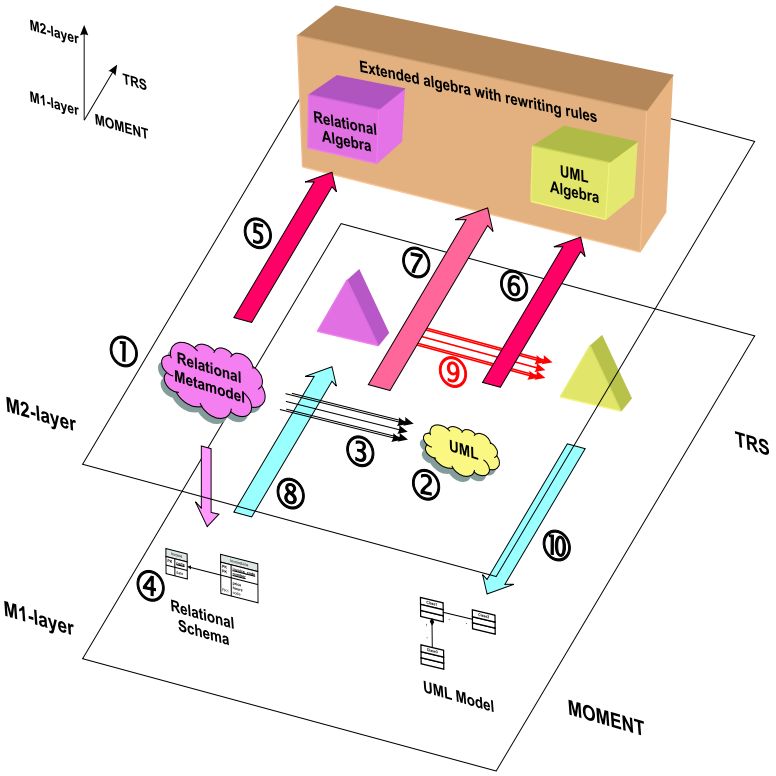


Fig. 3. Model Transformation

5.1 Overview of the MOMENT model transformation process

Transforming any model using the MOMENT Framework constitutes a process that is detailed in Fig. 3. To obtain the corresponding UML model from the relational schema of the motivating example, we perform the following steps:

(i) (1) and (2):
We specify both relational and UML metamodels, respectively, at the M2-layer of the MOMENT platform using the operations of the MOMENT algebra. Each one of the metamodels is a schema made up of concepts, which describe the main entities of the ontology, and by properties, which describine the concepts by specifying values and establishing relationships between them. These algebraic specifications are performed through visual wizards that are embedded in a specific CASE tool to disguise the equational logic formalism.

(ii) (3):
Mappings are specified between the concepts of both metamodels at the M2-layer by means of a script language, indicating semantic relationships. There are two kinds of equivalence mappings that can be expressed in this language:

(a) *Simple mappings*, which define a simple correspondence between two concepts that belong to different metamodels; for instance, between a table and a class, or between a column of a table and an attribute

of a class.

(b) *Complex mappings*, which define correspondences between elements of a source metamodel and a target metamodel. These mappings relate two structures of concepts that represent a similar semantic meaning. For instance, to define an equivalence relationship between a foreign key of the relational metamodel and an association of the UML metamodel, we have to relate the foreign key, the unique constraint and the not null value constraint concepts to the association concept. This is because all three of these concepts of the relational metamodel provide the necessary knowledge to define an association between two classes in the UML metamodel, such as the cardinalities of the association.

(iii) (4):

The original relational schema is specified by means of concepts and properties in a schema of the M1-layer of the MOMENT platform. Both concepts and properties are instances of the elements of the relational metamodel defined in step 1.

(iv) (5) and (6):

Both relational and UML metamodels, respectively, are compiled into algebraic theories by means of the *Theory Compiler* module of the Framework. The compilation uses the concepts to define the sorts of the new theory and the properties to define constructors and query operators. The generated theories are interpreted by the CafeOBJ TRS, providing the respective algebras to define models in the TRS as algebraic terms.

(v) (7):

The semantic mapping that is specified between the concepts of both metamodels at the M2-layer is also compiled into another theory that extends the theories described above with a set of rewriting rules. This theory indicates how to transform a model of the source metamodel (relational metamodel) into a new model of the target metamodel (UML) in an automatic way.

(vi) (8):

The original relational schema, which is defined in step (4) at the M1-layer of the MOMENT platform, is compiled into a term of the relational algebra in the CafeOBJ TRS by means of the *Term Manager* module of the Framework.

(vii) (9):

The TRS evaluates the term that represents the initial relational schema in the algebra obtained in step (7). The user can manage this process through the *Evaluator* module of the Framework. The evaluation process can be carried in a step-by-step mode or in only one step with the full-evaluation mode, benefitting from the evaluation features of the TRS. The TRS reduces the initial term by applying the rewriting rules obtained

in step (7), generating a term of the target algebra.

(viii) (10):

This is the last step of the model transformation process. It parses the obtained term in step (9), defining a model in the M1-layer as an instance of the target metamodel defined at the M2-layer. There, it is disguised with the visual metaphor associated to the target metamodel in the graphical CASE tool.

In the model transformation process, the user only interacts with the MOMENT platform when defining the source and target metamodels (step (1) and (2)), the semantic mappings between the elements of both metamodels (step (3)) and the initial model (step (4)). The other steps are automatically carried out by the Framework, although the user can participate in the evaluation process by specifying the rewriting rules to be applied by the TRS at each step of the term reduction.

In the following sections, we explain phases (5), (6), (7), (8) and (9) in more detail, indicating how the TRS is able to perform model transformations providing formal support to the objectives of MDA.

5.2 Compilation of equational logic based theories

The relational and UML metamodels defined at the M2-layer of the MOMENT platform are compiled into theories based on equational logic in steps (5) and (6), respectively. The compilation of MOMENT metamodels into equational theories uses the concepts of the metamodel to obtain the sorts of the theory; for instance, the sorts Table, Field, ForeignKey for the relational metamodel, as well as the identifiers for these sorts, i.e., the sorts TableId, FieldId and ForeignKeyId. The properties of a MOMENT metamodel provide information about the structure of the term of a sort by means of the cardinalities. Thus, when a concept A is related to a concept B by means of a property that has cardinality 1..1, the constructor of the sort A looks like this : $op\ a\_ : B \rightarrow A$. Nevertheless, if the minimum cardinality is zero or the maximum cardinality is *many*, then the constructor for a term of the sort A looks like this: $op\ a\_ : ListB \rightarrow A$, where $ListB$ is a *sort* that permits the definition of lists, whose items are terms of *sort B*. As CafeOBJ belongs to the OBJ language family, it permits equational specification through several equational theories, such as associativity, commutativity, identity, idempotence and combinations between all these. This feature is reflected at the execution level by term rewriting by means of such equational theories.

Fig. 4 shows the constructors of the compiled theory for the relational metamodel; and Fig. 5 shows the constructors for the UML metamodel. We obviate the definition of sorts and other constructors in the theory, as well as the definition of query operators, focusing on the elements of the metamodels that permit us to illustrate the example. We must point out that the constructors obtained for the UML theory permit us to define terms that represent

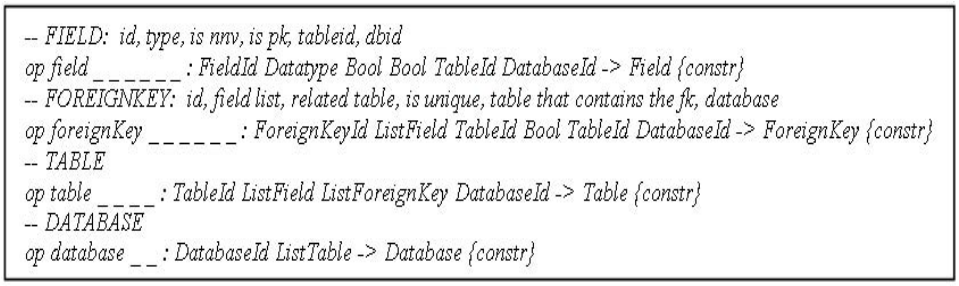


Fig. 4. Part of the relational theory

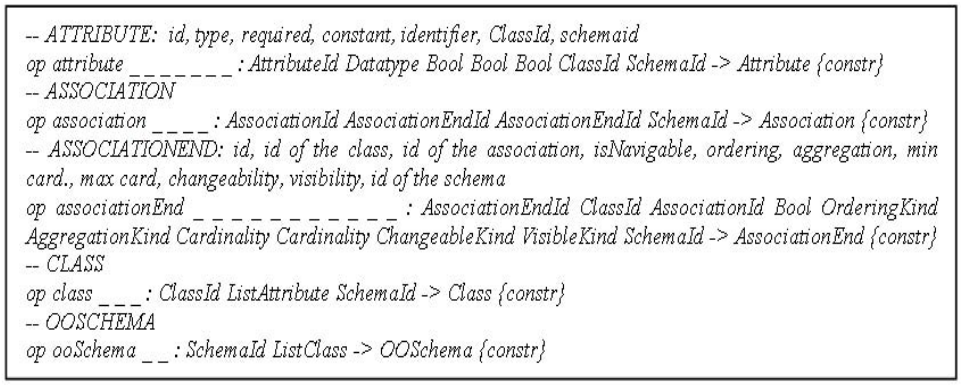


Fig. 5. Part of the UML theory

UML-compliant models.

5.3 Compilation of rewriting logic-based theories

To transform the relational schema of the example, semantic mappings are defined between the concepts of both source and target metamodels in step (3). These mappings are compiled into an algebra that extends both relational and UML algebras (steps (5) and (6)) with a set of rewriting rules describing the guidelines for the model transformation. These rules are automatically applied by the TRS rewriting the initial term into a term of the target algebra. The new algebra constitutes the context where semantical relationships between the source and target ontologies are defined. To allow the transformation process, the new algebra must relate the *sorts* of the initial algebra (relational metamodel) to the *sorts* of the target algebra (UML metamodel). Relationships between the sorts of both algebras result in a subsort order that involves all the sorts. For instance, in the example, the sort Table is a subsort of the sort Class, indicating that a class can take the place of a table that was there before. Subsort relationships affect all the sorts of both algebras, even identifiers and lists, because they are the related concepts in the MOMENT algebra.

The properties that relate concepts in the MOMENT algebra define a canonical order among the sorts of the compiled algebras. This order is taken

into account to generate the rewriting rules. We present the rewriting rules that are applied to the relational schema of the example to obtain a semantically equivalent UML model in CafeOBJ syntax:

(i) Field

A field of a table becomes an attribute of a class in the term that represents a UML model. The rule reuses the features of the field (datatype, whether it is null or not and whether it is primary key) to generate an attribute. Field features also indicate the attribute datatype, whether it is required or not and whether it is the identifier of the class to which it belongs:

op field _ _ _ _ _ _ : FieldId Datatype Bool Bool TableId DatabaseId -> Attribute
eq field FI D NNV PK TI DBI = attribute FI D NNV false PK TI DBI .

(ii) Foreign Key

A foreign key can define an association between two classes in the UML context. The following rule is applied when the foreign key is unique and not null, obtaining an association 1..1 - 0..* between the classes generated from the related tables.

op foreignKey _ _ _ _ _ _ : ForeignKeyId ListAttribute TableId Bool TableId DatabaseId -> ListClass
ceq foreignKey FkI LA RTI U TI DBI =
(association FkI TI RTI DBI)(associationEnd TI TI FkI true unordered aggregate card 0 many frozen public DBI)(associationEnd RTI RTI FkI true unordered none card 1 card 1 frozen public DBI)
if U and isRequired (LA) .

(iii) Table

A table becomes a class. The rewriting rules must take into account the fact that a table is made up of fields and foreign keys, so that a field will become an attribute of the new class and a foreign key will become a set of elements of the UML model, i.e., an association and two association ends, according to the UML metamodel.

op table _ _ _ _ : TableId ListAttribute ListClass DatabaseId -> ListClass
eq table TI LA nilForeignKey DBI = (class TI LA DBI) .
eq table TI LA LC DBI = (class TI LA DBI) LC .

(iv) Database

Finally, a database is rewritten into a term of the sort OOSchema, representing the target UML model, by means of the following rule:

op database _ _ : DatabaseId ListClass -> OOSchema
eq database DBI LC = ooSchema DBI LC .

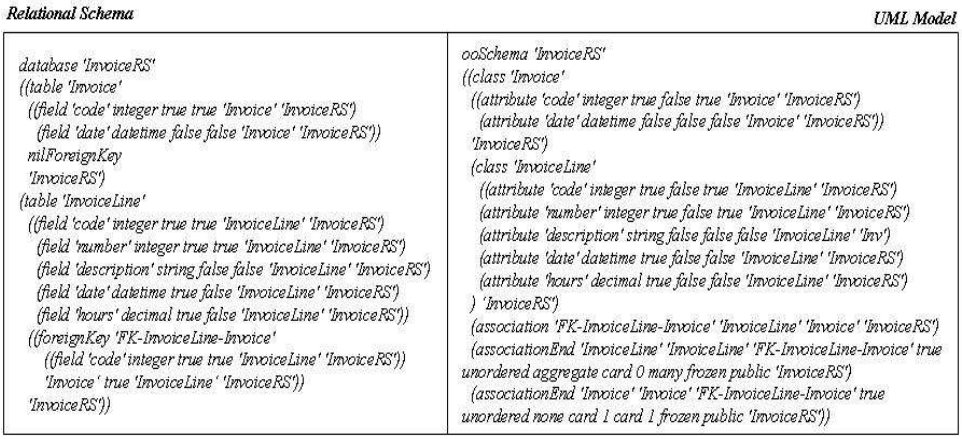


Fig. 6. Original term representing the source relational schema, and the generated term representing the target UML model

5.4 Term rewriting process

Step (8) compiles the relational schema defined in the M1-layer of the MOMENT platform, obtaining the algebraic term in Fig. 6.a. The TRS applies the rewriting rules specified above to this initial term, obtaining a term of the target algebra (UML), shown in Fig. 6.b. This term is parsed and is defined as a UML model in the M1-layer of the MOMENT platform. There, it is automatically related to graphical pictures in a specific CASE tool.

During the term rewriting process, some additional information could be required in order to perform a correct transformation, as the metamodels do not have the same expressive power. For instance when a transformation case has not been taken into account or when several rewriting rules can be applied to the source model. In this cases a visual wizard helps the user to chose one option or even to add a new transformation rule, providing a visual interface for the CafeOBJ interpreter.

To benefit from the MOMENT features, we have integrated the functionality of the MOMENT Framework into a visual modeling environment [22]. In this way, we can relate algebraic specifications to visual notations so that the user can use these graphics to build a model. The CASE tool we have chosen is MS Visio [23]. We have developed an add-in that permits the definition of metamodels with concepts and properties. Fig. 7 shows the interface that permits the definition of the graphical symbol of a class in the MOMENT platform.

In visual CASE tools, models are usually defined by dropping graphical primitives on a sheet where the model is defined. By means of the developed add-in, dropping a primitive on the sheet does not only add a figure to the model but it also defines it algebraically, specifying the model so that it can be manipulated afterwards.

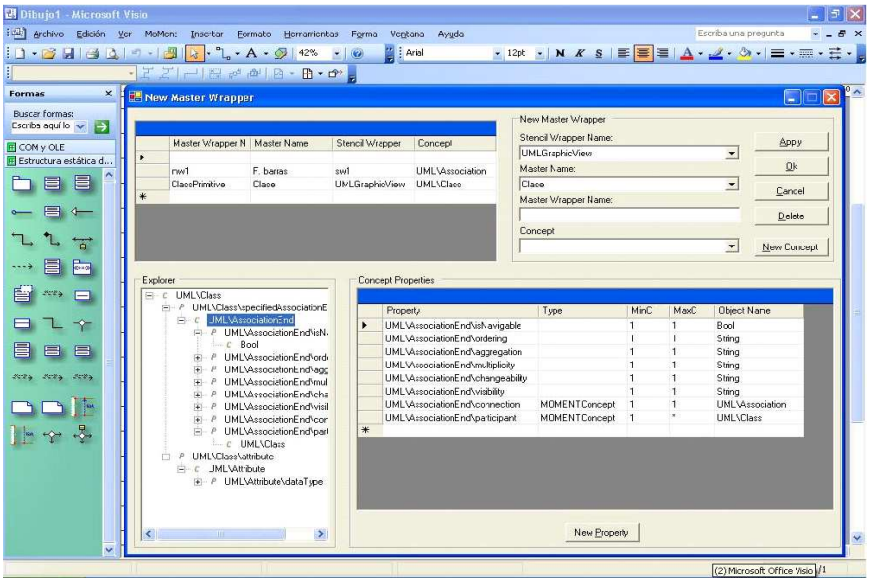


Fig. 7. Visual interface to define a graphical primitive algebraically

6 Conclusions and further work

Nowadays, software applications have become complex combinations of technology, which have to be well understood in order to manage them. The development of software artifacts involves models that can be mixed with others to obtain an entire system from partial views or that can be interconnected with others in order to guarantee both interoperability in a distributed environment and their implementations.

Model management [8] is an emerging research field whose aim is to resolve data model integration and interoperability by means of generic operators. Similarly, MDA raises the level of abstraction in the software development process by treating models as primary artifacts. MDA potentially covers the modeling of all aspects of a system throughout its life cycle, making software development processes easier and more automated.

The MOMENT (MOdel manageMENT) platform follows this trend by providing a framework where models can be represented using an algebraic approach. The MOMENT Framework benefits from the best features of current visual CASE tools and from the main advantages of formal environments such as term rewriting systems, combining both industrial and research features.

In this paper, we have presented the generic model transformation mechanism provided by the MOMENT Framework, focusing on the use of the CafeOBJ TRS to perform automatic translations of models. This mechanism has been applied to platform-independent models in the MDA context, obtaining a UML model from a relational schema. The functionality of TRSs permits us to deal with model management from a more abstract point of view, since the application of rewriting rules can be performed in a trans-

parent way. This fact allows us to focus all the efforts on the specification of models without having to take the evaluation logic into account. Our approach constitutes an algebraic baseline to cope with the future model transformation language QVT, providing a user-friendly environment to manipulate models from a visual CASE tool [22]. In [16], we present the fundamental mainstay on which we have built our MOMENT platform taking into account our previous experience in the industrial project RELS, a tool for the recovery of legacy systems.

Currently, we are working with transformations between relational schemas and UML models. In the near future, we will also take into account software architecture specifications by means of PRISMA ADL [17], studying semantic interoperability between software architectures and other types of software artifacts represented through UML models.

7 Acknowledgments

This work was supported by the Spanish Government under the National Program for Research, Development and Innovation, DYNAMICA Project TIC 2003-07804-C05-01.

References

[1] S. Cook, *Domain-Specific Modeling and Model Driven Architecture*, MDA Journal, (January 2004).

[2] S. Melnik, E. Rahm, P. A. Bernstein, *Rondo: A Programming Platform for Generic Model Management*, (Extended Version), Technical Report, Leipzig University, 2003. Available at http://dol.uni-leipzig.de/pub/2003-3.

[3] OMG, *The Model-Driven Architecture*, Guide Version 1.0.1, OMG Document: omg/2003-06-01. Available from www.omg.org.

[4] OMG, *Meta Object Facility 1.4*, OMG Document: formal/02-04-03. Available from www.omg.org.

[5] OMG, *MOF 2.0 Query/Views/Transformations RFP*, OMG Document ad/2002-04-10. Available from www.omg.org.

[6] Alagic, S. and Bernstein, P.A., *A Model Theory for Generic Schema Management*, in Proceedings of DBPL'01, G. Ghelli and G. Grahne (eds), Springer-Verlag, (2001).

[7] Madhavan, J., P.A. Bernstein, and E. Rahm, *Generic Schema Matching using Cupid*, MSR Tech. Report MSR-TR-2001-58, 2001, http://www.research.microsoft.com/pubs (short version in VLDB 2001).

[8] Bernstein, P.A., Levy, A.Y., Pottinger, R.A., *A Vision for Management of Complex Models*, Microsoft Research Technical Report MSR-TR-2000-53, June 2000, (short version in SIGMOD Record 29, 4 (Dec. '00)).

[9] N. Revault, H.A. Sahraoui, G. Blain and J.F. Perrot, *A Metamodeling technique: The MÉTAGEN system*, TOOLS 16: TOOLS Europe'95, Prentice Hall, pp. 127-139. Versailles, France. Mar. 1995.

[10] N. Revault, X. Blanc and J. F. Perrot, *On Meta-Modeling Formalisms and Rule-Based Model Transforms*, Comm. at workshop, In Iwme'00 workshop at Ecoop'00, Jean Bézivin and Johannes Ernst (ed), Sophia Antipolis and Cannes, France, June, 2000.

[11] ISO/IEC 10746-1, 2, 3, 4 — ITU-T Recommendation X.901, X.902, X.903, X.904, *Open Distributed Processing - Reference Model*. OMG, 1995-96.

[12] CDIF Technical Committee, *CDIF Framework for Modeling and Extensibility*, Electronic Industries Association, EIA/IS-107, January 1994. See http://www.cdif.org/.

[13] Razvan Diaconescu, Kokichi Futatsugi, *An overview of CafeOBJ*. Electronic Notes in Theoretical Computer Science vol. 15: (2000)

[14] Microsoft Research (Don Syme), *The F# official web site*: http://research.microsoft.com/projects/ilx/fsharp.aspx.

[15] N. Martí-Oliet and J. Meseguer, *Rewriting logic: roadmap and bibliography*, Theoretical Computer Science, vol. 285, issue 2 (August 2002), pp. 121-154. Preprint version available at http://maude.cs.uiuc.edu.

[16] A. Boronat, J. Pérez, J. Á. Carsí, I. Ramos, *Two experiencies in software dynamics*. Journal of Universal Science Computer. Special issue on Breakthroughs and Challenges in Software Engineering. April 2004.

[17] J. Pérez, I. Ramos , J. Jaén, P. Letelier, E. Navarro , *PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures*, 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, November 6 - 7, 2003 IEEE Computer Society Press pp. 59-66.

[18] Elliotte Rusty Harold, *XML Bible*, IDG Books Worldwide, 1999.

[19] OMG, *XML Metadata Interchange (XMI) Specification*, OMG Document formal/02-01-01.

[20] W3C, *XSL Transformations (XSLT) v1.0. W3C Recommendation*, http://www.w3.org/TR/xslt, Nov. 1999.

[21] M. Peltier, J. Bézivin, and G. Guillaume, *MTRANS: A general framework, based on XSLT, for model transformations*. In WTUML01, Proceedsings of the Workshop on Transformations in UML, Genova, Italy, Apr. 2001.

[22] Artur Boronat, José Á. Carsí, Isidro Ramos, Julián Pedrós, *Soporte Formal para Entornos Visuales de Modelado*, I Workshop Metodologies for Dynamic User Interfaces Development. Ed. Computer Science Department. Castilla-La-Mancha University. Albacete (Spain), July, 2004. (In Spanish).

[23] Graham Wideman, *Microsoft Visio 2003 Developer's Survival Pack*, Trafford, 2004.

Evolution of Language Interpreters

Ralf Lämmel [1]

Vrije Universiteit Amsterdam & Centrum voor Wiskunde en Informatica

Abstract

We are interested in evolution scenarios for language-based functionality. We identify different dimensions along which such functionality can evolve, including the following: (i) coding style; (ii) coding details; (iii) data model; (iv) crosscutting concerns; and (v) patches. We focus at language interpreters as examples of language-based functionality, but similar scenarios exist for type checkers, static analyses, program transformations, and other sorts of language-based functionality. Our experiences are based on using rule-based programming (with Prolog) for the implementation of language-based functionality, while evolutionary transformations of the functionality are perceived as meta-programs.

Keywords: Evolution, Evolutionary Transformations, Rule-Based Programming, Language Interpreters, Language-Based Functionality, Meta-Programming, Prolog

Note: This extended abstract extracts material from these publications: "Evolution of rule-based programs" [5] and "Evolution scenarios for rule-based implementations of language-based functionality" [6]. The accompanying proof-of-concept implementation is the Prolog-based Rule Evolution Kit [12] (REK).

Introduction

Language-based tools involve some (ad-hoc) elements of a language's intended semantics. The amount of adopted semantics depends on the specific service that is provided by a tool. Here are examples. An analysis tool typically implements a so-called abstract interpretation, which rephrases the normal semantics in terms of abstract domains of meanings. A transformation tool supposedly resembles the intended semantics in so far that it employs algebraic laws as well as typing and scoping rules. A translation tool implements a syntax-to-syntax mapping, which ideally can be complemented into a commuting diagram with nodes for the two syntaxes and the two semantics of the involved languages.

[1] Email: ralf.laemmel@cwi.nl

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Type checkers and language interpreters can be seen as prototypical examples of language-based tools. Such tools implement the language semantics rather directly. The extensibility of such semantics-oriented programs (and the underlying formal semantics descriptions) has received ample attention in the programming-language community. There exist highly advanced approaches to the reuse of language descriptions or components thereof, e.g., monad-style denotational semantics [8], action semantics [9], abstract state machines [2], strategic programming [7], modular SOS [10], and modular attribute grammars [3]. This suggests that the domain of language descriptions is suitable for studying evolution of language-based tools in general.

In our work, we aim at a simple, pragmatic, re-engineering-like approach to the evolution of language-based tools. We use automated transformations in the sense of meta-programming for the operationalisation of evolution scenarios. We will focus here on language interpreters. We assume that language interpreters are programmed in a rule-based language, say in Prolog. We employ a suitably designed operator suite for evolutionary transformations. These operators support the following activities:

- Restructuring to prepare for extensions or revisions.
- Extension to add new concerns by modular composition or weaving.
- Revisions to remove or to change inappropriate parts.

Dimensions of evolution

We are going to work through some evolution scenarios for an interpreter of a simple expression-oriented language. Let us assume that the interpreter is defined by a Prolog predicate `evaluate` of the following type:

```
:- profile evaluate(+exp,+varenv,-val).
```

REK supports typeful Prolog programming on the basis of profiles as shown above. The sort `exp` corresponds to the syntactical domain of expression forms. The sort `varenv` models variable environments. The sort `val` models the type of evaluation results, which are numbers so far. That is:

```
:- data  exp    = const(number) | var(varid) | ... .  % expression forms
:- alias varid  = atom.          % variable identifiers
:- alias varenv = [(varid,val)]  % variable environments
:- alias val    = number.        % evaluation results
```

There is one Prolog clause per expression form. We omit these rules.

Evolution in the sense of style conversion

The type that we gave for the predicate `evaluate` above implies big-step style. Some language extensions are more easily accommodated when small-step style is chosen, e.g., exception handling or concurrency are of that kind. A typical evolution step is then to convert an interpreter from big-step to small-step style. We note that there are further kinds of conversion that relate to

evolution, e.g., CPS conversion.

To prepare the small-step to big-step conversion, we need to enable small steps as far as the type of `evaluate` is concerned. That is, rather than returning a value of sort `val`, we must return an expression of sort `exp`, which is potentially subject to further reduction. To this end, all rules need to be adapted such that values are injected into expressions via a dedicated functor. REK provides a transformation operator `othertype` which does just that. The adapted type of the predicate `evaluate` is this:

```
:- profile evaluate(+exp,+varenv,-exp).
```

The intrinsic part of the conversion is about rule shredding. That is, we need to take apart big-step rules such that many small-steps rules are obtained. REK provides a dedicated transformation operator `big2small`.

Evolution with regard to the data model

Let us assume that we want to accommodate object-oriented constructs. It is straightforward to extend the syntactical domain `exp` with expression forms for method calls, field access, object construction, and others. However, the interpretation of these constructs cannot be accomplished by just adding rules to the original predicate `evaluate`.

One problem is that the type of evaluation results needs to be made fit such that different kinds of results are accommodated. In fact, in our running example, the definition of `val`, as a type alias, is in the way. We need to turn `val` into a proper datatype with one alternative for numbers, and another one for object references. REK provides an operator `othertype`, which supports such evolution in the data dimension. The adapted sort `val` looks as follows:

```
:- data val = num(number)    % results as before
            | oref(integer). % new kind of result
```

Clearly, the `othertype` operator does not just adapt the sort `val`, but also the actual interpreter rules. That is, all pre-existing positions of type `val` have to wrap numbers with the functor `num`.

Evolution with regard to crosscutting concerns

The object-oriented language extension also requires an enhanced predicate for expression evaluation. So far, we only pass around an environment for variables. We need to add parameters for a virtual method table (cf. sort `vmt` below), for the current object (cf. sort `this`), and for an object store (cf. sort `store`). The original rules have to be adapted such that they participate in a data flow for these new semantic components.

Such data flow or computation that affects many or all existing rules is best viewed as the implementation of a crosscutting concern in the sense of aspect-oriented programming. (This link between aspect-oriented programming, rule-based programming and program transformation is explored in [4].)

For comparison, here is the original type of the predicate `evaluate`:

```
:- profile evaluate(+exp,+varenv,-val).
```

The OO-enabled predicate must be of the following type:

```
:- profile evaluate(+exp,+varenv,+vmt,+this,+store,-val,-store).
```

REK provides operators for adding positions and establishing the appropriate data flow as needed above. The operator `add` enhances the type of a predicate, and it adds fresh variables to the relevant literals in the rules. The operator `thread` adapts rules such that all relevant variables are unified in a way to encode the intended data flow.

Evolution in the sense of conservative extension

We are now in the position to add the rules for the object-oriented constructs. The previous advances of the data model and data flow have made it possible to perform a truly conservative extension in the end: the rules to be added do not affect the reduction of programs that do not refer to the new constructs [1]. This sort of evolution is very simple because it basically means to 'put together' two rule sets as opposed to an invasive transformation of rules.

Evolution in the sense of point-wise restructuring

Rule-based programs can also be subjected to rather specific restructuring transformations, where the programmer points out locations of interest. As an illustration, we will improve one particular detail of the interpreter that we obtained so far. That is, we are going to reduce the number of arguments of the predicate for expression evaluation.

For comparison, the current profile is this:

```
:- profile evaluate(+exp,+varenv,+vmt,+this,+store,-val,-store).
```

It seems that having three positions `+varenv,+vmt,+this` is somewhat outrageous since these positions are all concerned with environment-like information. All this information is passed on to subcomputations.

So we aim at *compound* environments with the following structure:

```
:- alias env = (varenv,vmt,this).
```

The profile of the predicate for evaluation is simplified as follows:

```
:- profile evaluate(+exp,+env,+store,-val,-store).
```

The required grouping effort is simply automated by REK's `group` operator. Again, this operator does not simply change the predicate type, but the grouping also affects all relevant literals in the many rules.

The illustrated grouping transformation operates at the level of predicate positions. One can also consider forms of restructuring that operate at other levels, e.g., the level of functor positions or the the level of rule bodies. Folk-

lore examples of transformations at the level of rule bodies are folding and unfolding, where unfolding means to symbolically perform predicate application, and folding is the inverse [11]. More generally, all kinds of refactoring transformations can be instantiated for language interpreters and other rule-based programs.

Evolution in the sense of patching

We consider one more evolution scenario. Let us assume that we want to enable logging of method calls. Thereby, we would obtain a simple debugging facility for the interpreted object-oriented language. It is relatively straightforward to adapt the OO interpreter for this purpose. We basically need to adapt the interpreter rule for method calls such that method calls are logged. Rather than messing with existing rules, we can apply an evolutionary transformation that records the intent of adaptation more explicitly and separately. REK offers a corresponding `inject` operator, which allows one to enhance the body of a given rule by stating the additional literals.

Concluding remarks

We have sketched some evolution scenarios for language interpreters. (For a profound presentation, we refer to [5,6].) This work contributes to the emerging field 'software evolution for language-based functionality'. The increasing interest in this field is motivated by new applications related to modelling and meta-modelling in software development. These applications call for better understanding of the evolution of languages (or meta-models) and language-based functionality (or model-driven transformations).

Our transformational approach to evolution receives input from other disciplines such as program and data refinement, program synthesis, transformational program development (from specifications), data re-engineering, and grammarware engineering. We have deployed evolutionary transformations as a general method for restructuring, extending, shrinking and revising language-based functionality such as interpreters.

Future work on the subject has to provide practically useful tool support for the evolution of rule-based programs, a comprehensive analysis of basic and composed evolution operators, and a meaningful, formal model of evolution, with coverage of transformations that are not strictly semantics-preserving.

References

[1] Aceto, L., W. Fokkink and C. Verhoef, *Conservative extension in structural operational semantics*, Bulletin of the European Association for Theoretical Computer Science **69** (1999), pp. 110–132, columns: Concurrency.

[2] Börger, E. and R. Stärk, "Abstract State Machines: A Method for High-Level System Design and Analysis," Springer-Verlag, 2003.

[3] Kastens, U. and W. Waite, *Modularity and reusability in attribute grammars*, Acta Informatica 31 (1994), pp. 601–627.

[4] Lämmel, R., *Declarative aspect-oriented programming*, in: O. Danvy, editor, *Proceedings PEPM'99, 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM'99, San Antonio (Texas), BRICS Notes Series NS-99-1*, 1999, pp. 131–146.

[5] Lämmel, R., *Evolution of Rule-Based Programs*, Journal of Logic and Algebraic Programming **60–61C** (2004), pp. 141–193, in: Special Issue on Structural Operational Semantics, edited by L. Aceto and W. Fokkink.

[6] Lämmel, R., *Evolution scenarios for rule-based implementations of language-based functionality*, in: L. Aceto, W. Fokkink and I. Ulidowski, editors, *Proc. of SOS workshop (Structured Operational Semantics)*, ENTCS (2004), 20 pages. To appear.

[7] Lämmel, R., E. Visser and J. Visser, *The Essence of Strategic Programming* (2002–2004), draft; Available at http://www.cwi.nl/~ralf.

[8] Moggi, E., *Notions of computation and monads*, Information and Computation **93** (1991), pp. 55–92.

[9] Mosses, P., "Action Semantics," Number 26 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1992.

[10] Mosses, P., *Pragmatics of Modular SOS*, in: H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology: 9th International Conference, AMAST'02*, LNCS **2422** (2002), pp. 21–40.

[11] Pettorossi, A. and M. Proietti, *Rules and Strategies for Transforming Functional and Logic Programs*, ACM Computing Surveys **28** (1996), pp. 360–414.

[12] *The Rule Evolution Kit — downloads, documentation, papers, on-line demos* (2004), maintained by R. Lämmel; http://www.cs.vu.nl/rek/; Version of the software: 0.78.

Reflective Designs

Robert Hirschfeld [1]

DoCoMo Communications Laboratories Europe
Munich, Germany

Ralf Lämmel [2]

Vrije Universiteit & Centrum voor Wiskunde en Informatica
Amsterdam, The Netherlands

Abstract

We render runtime system adaptations by design-level concepts such that running systems can be adapted and examined at a higher level of abstraction. The overall idea is to express design decisions as applications of design operators to be carried out at runtime. Design operators can implement design patterns for use at runtime. Applications of design operators are made explicit as design elements in the running system such that they can be traced, reconfigured, and made undone.

Our approach enables Reflective Designs: on one side, design operators employ reflection to perform runtime adaptations; on the other side, design elements provide an additional reflection protocol to examine and configure performed adaptations. Our approach helps understanding the development and the maintenance of the class of software systems that cannot tolerate downtime or frequent shutdown-revise-startup cycles.

We have accumulated a class library for programming with Reflective Designs in Squeak/Smalltalk. This library employs reflection and dynamic aspect-oriented programming. We have also implemented tool support for navigating in a system that is adapted continuously at runtime.

Keywords: Reflective Designs, Runtime Adaptation, Design Elements, Design Operators, Design Patterns, Reflection, Method-Call Interception, Meta-Programming, Aspect-Oriented Programming, Dynamic Weaving, Dynamic Composition, AspectS, Squeak, Smalltalk, Metaobject Protocol

Note: This extended abstract summarises our full paper [7].

[1] Email: hirschfeld@docomolab-euro.com
[2] Email: ralf.laemmel@cwi.nl

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Runtime system adaptation

Our work on Reflective Designs is concerned with adaptation of software systems at runtime, as needed for dynamic component coordination [11], runtime system configuration [1], dynamic service adaptation [5,6], and rapid prototyping without shutdown-revise-startup cycles [12]. Runtime adaptability is crucial for systems with strong availability demands, such as in telecommunications. Downtime of such systems can barely be tolerated. Software maintenance and evolution has to be carried out in the running system.

Reflective Designs enhance object-oriented design and programming by techniques for runtime system adaptation. There are two key notions: *design elements* and *design operators*, which we will explain in turn.

Design elements

We contend that a program is structured according to design decisions. We require that design decisions are represented explicitly in the program. Thereby, software design will be traceable in the program. We even require that design decisions are to be represented explicitly in the running system. We use the term design element to denote representations of design decisions in programs. In fact, we require that design elements are amenable to reflection such that design decisions can be observed and modified at runtime. With that, the notion of runtime system adaptations boils down to explicit construction, modification, and retirement of design elements.

Design elements can be examined and (re-) configured. Here, examination and (re-) configuration are used in the sense of introspection and intercession. Normal object-oriented introspection and intercession concerns the fields and methods of objects. Design-level introspection and intercession concerns design-level concepts such as the *participants* for a given design element. The examination of participants exemplifies design-level introspection. The configuration of participants and their roles exemplifies design-level intercession. Furthermore, for each object in the running system, we can introspect effective adaptations, i.e., the list of design elements that affect the object at hand.

Design operators

When compared to basic techniques such as the use of a metaobject protocol [8], the use of design elements makes runtime system adaptations more disciplined and more manageable. To this end, we provide abstractions that capture common design elements in a reusable manner. Applications of such abstractions perform system adaptations at a design level; hence, we call them design operators. Our work, so far, has concentrated on operators that model the realisation of common design patterns. The view 'design patterns as operators' also occurs in previous work [15,2,9,10,13,14]. The novelty of our

work is that our operators serve for *runtime* system adaptation, and *runtime* reflection on designs.

We can distinguish at least three kinds of operators. Additive operators superimpose additional structure or behaviour onto the running software system. Subtractive operators define and remove slices of behaviour or structure in the running software system. Refactoring operators revise the running system in a semantics-preserving manner.

It is clear that design operators can only be provided in the context of a sufficiently reflective programming system. Actual applications of design operators result in two effects. Firstly, the corresponding design elements are constructed. Secondly, the system's actual structure and behaviour is adapted as intended by the underlying design decision. Applications of design operators can be made undone by deactivating the corresponding design element. In case an inactive element is never ever needed again, we can let the element retire.

Implementation in Squeak/Smalltalk

We have developed the Reflective Designs framework as a class library for Squeak/Smalltalk. The implementation makes original use of infrastructure for reflection, method wrappers [3], and dynamic aspect-oriented programming with AspectS [4]. The Reflective Designs framework involves several layers of abstraction, while these layers are presented as APIs to the programmer. The idea is that layers at a higher level of abstraction perform less lower level reflection. Using the Reflective Designs framework, we have exercised some scenarios of runtime system adaptations.

We have also provided interactive tool support for reflective designs. Accordingly, we have extended some existing tools, such as the normal system browser, and we have provided new tools such as a dedicated 'reflective designs center'. The tool extensions are particularly interesting in so far that we have implemented them as self-applications of the Reflective Designs framework, e.g., the system browser is adapted by appropriate design elements.

Concluding remarks

We believe that Reflective Designs and our prototypical implementation of this approach provide useful input for further research on runtime system adaptation. Major directions for future work are the following. Firstly, the fusion of Reflective Designs and refactoring transformations should be completed. We note that we have focused on additive and subtractive adaptations in our work so far. Secondly, the robustness of Reflective Designs should be improved by dedicated system analyses and rollback mechanisms. Thirdly, our practical approach to reflective designs needs to be complemented by formal support. The ultimate goal is an approach where runtime system adaptations are as powerful and robust as static meta-programs today.

References

[1] Akkawi, F., A. Bader and T. Elrad, *Dynamic Weaving for Building Reconfigurable Software Systems* (2001), in Online Proc. of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems.

[2] Aßmann, U., *AOP with design patterns as meta-programming operators*, Technical Report 28, Universität Karlsruhe (1997).

[3] Brant, J., B. Foote, R. Johnson and D. Roberts, *Wrappers to the Rescue*, in: E. Jul, editor, *ECOOP '98*, LNCS **1445** (1998), pp. 396–417.

[4] Hirschfeld, R., *AspectS – Aspect-Oriented Programming with Squeak*, in: M. Aksit, M. Mezini and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, LNCS **2591** (2003), pp. 216–232.

[5] Hirschfeld, R. and K. Kawamur, *Dynamic Service Adaptation*, in: *Proc. DARES (ICDCSW'04)*, Hachioji, Tokyo, Japan, 2004, pp. 290–297.

[6] Hirschfeld, R., K. Kawamura and H. Berndt, *Dynamic Service Adaptation for Runtime System Extensions*, in: R. Battiti, M. Conti and R. Lo Cigno, editors, *Wireless On-Demand Network Systems*, LNCS **2928** (2004), pp. 227–240.

[7] Hirschfeld, R. and R. Lämmel, *Reflective Designs*, IEE Proceedings Software (2004), Special Issue on Reusable Software Libraries. To appear. Available at http://homepages.cwi.nl/~ralf/rd/.

[8] Kiczales, G., J. des Rivieres and D. Bobrow, "The Art of the Metaobject Protocol," MIT Press, Cambridge, MA, USA, 1991, viii + 335 pp.

[9] Krishnamurthi, S., Y.-D. Erlich and M. Felleisen, *Expressing Structural Properties as Language Constructs*, in: S. Swierstra, editor, *Proc. ESOP'99*, number 1576 in LNCS, 1999, pp. 258–272.

[10] Ludwig, A., "Automatische Transformation großer Softwaresysteme," Ph.D. thesis, Universität Karlsruhe (2002).

[11] Pinto, M., L. Fuentes, M. Fayad and J. Troya, *Separation of Coordination in a Dynamic Aspect Oriented Framework*, in: *Proc. AOSD'02* (2002), pp. 134–140.

[12] Popovici, A., T. Gross and G. Alonso, *Dynamic Weaving for Aspect Oriented Programming*, in: *Proc. AOSD'02* (2002), pp. 141–147.

[13] Sullivan, G., *Advanced Programming Language Features for Executable Design Patterns*, Technical Report AIM-2002-005, MIT AI Laboratory (2002).

[14] von Dincklage, D., *Making Patterns Explicit With Metaprogramming*, in: *Proc. GPCE'03*, LNCS (2003), pp. 287–306.

[15] Zimmer, W., "Frameworks und Entwurfsmuster," Ph.D. thesis, Universität Karlsruhe (1997).

Towards a Megamodel to Model Software Evolution Through Transformations

Jean-Marie Favre[⋆] Tam NGuyen

Laboratoire LSR-IMAG
University of Grenoble, France
http://www-adele.imag.fr/∼jmfavre

Abstract

Model Driven Engineering is a promizing approach that could lead to the emergence of a new paradigm for software evolution, namely Model Driven Software Evolution. Models, Metamodels and Transformations are the cornerstones of this approach. Combining these concepts leads to very complex structures which revealed to be very difficult to understand especially when different technological spaces are considered such as XMLWare (the technology based on XML), Grammarware and BNF, Modelware and UML, Dataware and SQL, etc. The concepts of model, metamodel and transformation are usually ill-defined in industrial standards like the MDA or XML. This paper provides a conceptual framework, called a megamodel, that aims at modelling large-scale software evolution processes. Such processes are modeled as graphs of systems linked with well-defined set of relations such as *RepresentationOf* (μ), *ConformsTo* (χ) and *IsTransformedIn* (τ).

Key words: model driven engineering, meta-model, software evolution, mda, megamodel

1 Introduction

Model Driven Engineering (MDE) is a promising approach to develop and evolve software. Model, Metamodel and Transformations are the basic concepts of MDE. These concepts are far from new. They were already used in Ancient Egypt, though there were not formalized as such [3]. More recently, these concepts have been studied in many fields of Computer Science, may be under different perspectives and using other terminology. The *Model Driven Architecture* (MDA) standard, launched by the OMG in 2001 [14], had just popularized the vision that models, metamodels and transformations could play a central role in software engineering.

⋆ jmfavre@imag.fr

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The OMG gran vision presenting the MDA as the next paradigm in software engineering [14] is a source of strong debate. MDA is poorly defined, too complex, restrictive with the imposed use of MOF standard [14]. More importantly previous, yet similar approaches, such as syntax-driven approaches, have failed to find their path in industry. In fact we believe that there is nothing new in MDA, but thats why this may work this time (Episode I[6],). MDA could be more successful than previous because the software engineering community is more mature, concepts are better understood and tools are already there.

1.1 MDE and Technological Spaces

MDE is not MDA however. In fact, MDA is just a specific incarnation of the Model Driven Engineering approach which is applied to software systems. MDE is by no means restricted to the MDA standard. In fact, the MDE approach might not be restricted to the development and evolution of software systems, though this is on what we concentrate. MDA is a complex set of technologies dominated by the MOF god (Episode II[7]). MDE is on the contrary an open and integrative approach that embraces many other *Technological Spaces* (TSs) in a uniform way [13]. In this paper, the focus is on Software Technological Spaces, that is those used to produce software. The emphasis of MDE is on bridges between technological spaces, and on integration of bodies of knowledge developed by different research communities. Examples of TSs include not only MDA and MOF, but also Grammarware [12] and BNF, Documentware and XML, Dataware and SQL, Modelware and UML, etc. In each space, the concepts of model, metamodel and transformation take a different incarnation. For instance what is called a "metamodel" in Modelware corresponds to what is called a "schema" in Documentware and Dataware, a "grammar" in Grammarware, or even a "viewpoint" in the software architecture community [9]. In fact the concept of model, metamodels, and transformation are poorly defined in MDA, and this is the same in other standards such as XML. The true essence of these concepts is deeply buried into complex technologies.

1.2 Modelling software evolution

Getting a better understanding of these concepts is important, in particular to model software evolution. The focus of this paper is not on small scale software. These software products can be evolved without problem in an ad-hoc way. We are on the contrary interested in *evolution-in-the-large*, that is the evolution of large-scale industrial software systems. The evolution of these systems often involve various Technologicial Spaces over time, and various TSs are usually used at the same time. Whatever the technology used, recognizing the concepts of model and metamodels is important in this context [5]. In particular these concepts explain the metamodel/model

co-evolution phenomenona. The notion of model itself is also required to understand model/code co-evolution.These problems are not theoretical. They correspond to actual issues with strong implication on software industry development processes.

1.3 Towards a megamodel for MDE

Following the series "From Ancient Egypt to Model Driven Engineering" [3], the goal of this paper is to provide a *megamodel* that is "good enough" to describe MDE. Simply put this "megamodel", which is a model of MDE, should explain what is a model, a metamodel, a transformation, but also what is a transformation model, a model transformation, a model of model, a metamodel of transformation, and any combination of these terms. The megamodel should make it possible to reason about a complex software engineering process without entering into the details of technological space involved. Obviously the results obtained when reasoning on the megamodel must be consistent with those that would be obtained directly with the reality. Technically this megamodel is a metamodel, and therefore a model [3]. But since these terms are defined by the megamodel, calling it a metamodel would be confusing.

The goal of this paper is by no means to invent new concepts. On the contrary we just want to model what already exist. Nothing more. Instead of defining new words this paper relies on existing research on MDE [17][2][11][10][1]. In [17], Seidewitz describes informally, yet thoughtfully, models and meta-models. Bézivin identifies two fundamental relations coined *RepresentationOf* and *ConformsTo* [2]. Atkinson and Kuhne study the relationship between MDA and ontologies [1]. Almost all pieces of work carried out to define MDE concepts are either very specific and restricted to a particular TS, or they are expressed in plain english. By contrast the mega-model presented in this paper is expressed in UML with OCL constraints.

This paper presents the current version of the megamodel we have built so far. This megamodel has been carefully designed, and more importantly it has been validated through a large number of examples from different technological spaces. In [3], the study of MDE is taken from an historical perspective and it is shown how artefacts from Ancient Egypt to modern software technologies all conform to the megamodel in a smooth way.

The megamodel is summarized in Figure 10 at the end of this paper. It is made of 5 core associations, namely δ, μ, ϵ, χ and τ. It describes the concepts of model, language, metamodel, and transformation. The reader is invited to refer to the series "From Ancient Egypt to Model Driven Engineering" in which, each association is described in a different episode with plenty of concrete examples. For instance Episode I [6] concentrates on *models* and μ. Episode II [7]concentrates on *languages* and *metamodels*, that is ϵ and χ. Other episodes are under construction.

1.4 Structure of the paper

The remainder of the paper is structured as following. The basics of the megamodel are presented in Section 2. Transformations and *IsTransformedIn* (τ) are then introduced in Section 3. Finally Section 4 shows first results in modelling evolution and Section 5 concludes the paper.

2 Models, Languages, and Metamodels

As shown in the next UML class diagram, the core of MDE megamodel is centered around four relations: δ, μ, ϵ, and χ (Figure 2). Each relation is briefly discussed below in a separate section. For further information about models and μ, refer to Episode I [6]; for languages, metamodels, ϵ and χ, please refer to Episode II [7].

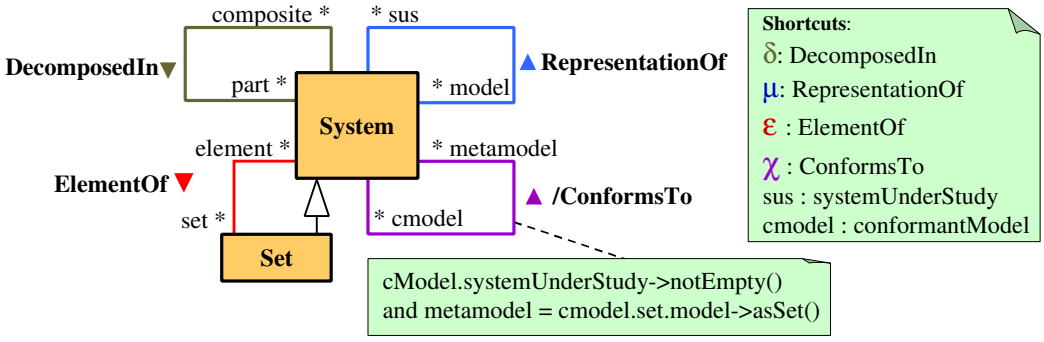


Fig. 1. MegaModel: δ, μ, ϵ, χ

2.1 Systems, Parts and DecomposedIn (δ)

A system is the primary element of discourse when talking about MDE.

This very abstract definition is just here to ensure a broad application of the megamodel. In short everything is a system, yet the use of the term "system" is not really important. Systems can be very simple. For instance the trigonometric value π is a system. The pair $(0000011, 0001101)$ is also a system. Complex system can be decomposed in subsystems or *parts*, leading to the definition of the *DecomposedIn* relation (δ) (Figure 2).

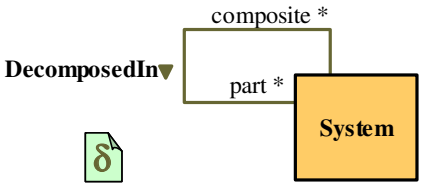


Fig. 2. MegaModel: δ

For instance $(0011, 1101)$ δ 1101 just indicates that the second system is a "part" of the first system. This information could be represented as a δ link

in a UML object diagram, but to save space, we prefer in this paper to use the traditional xRy mathematical notation, which is a shortcut to $(x, y) \in R$. Remember that relations are simply set of pairs in the set theory.

2.2 Models and RepresentationOf (μ)

Instead of providing yet-another definition of what a model is, lets cite existing definitions.

"A model is an abstraction of a physical system, with a certain purpose." (UML Std). "A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system." [10]. "A model is a set of statements about some system under study (SUS)." [15].

From these definitions we can at least identify three notions: the notion of *model*, the notion of *system under study* (SUS) and a relationship between these notions. This relation is called *RepresentationOf* in [2], so we kept the same terminology. We just use μ as shortcut to avoid wrong connotations and misinterpretations. The μ association is depicted in Figure 3.Episode I [6] is dedicated to the study of this association.

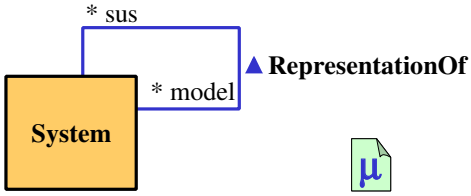


Fig. 3. MegaModel: μ

Lets just summarize here the main properties of this relation. It is key to recognize that the notion of model is relative. This is not an intrinsic property of a system. For instance, $(0011, 1101)$ is a system and it could be just a system. But one can state that this system plays the role of model by arguing that $(0011, 1101)\ \mu\ \pi$. One can indeed interpret the two parts of this pair as sequence of bits, and the result as a decimal representation of this the 3.14 value. We can state $(0011, 1101)\ \mu\ (3, 14)$ and $(3, 14)\ \mu\ \pi$. This example shows that μ links can be combined. The combination of μ and δ links leads to the notion of *interpretation* which is well explained in [17].

2.3 Languages, Sets, and ElementOf (ϵ)

In the language theory a *language* is defined as a *set* of sentences. For instance the set $\{"h", "ho", "hoo", "hooo", ...\}$ is the language of words that start with an h and continue with o letters. Lets call this set hoL. The language theory is built on the set theory. In the megamodel, the association *ElementOf* (ϵ) models this concept (see Figure 4).

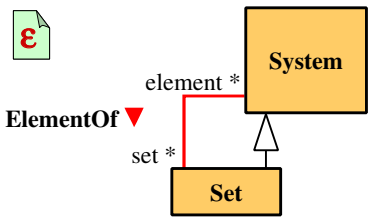


Fig. 4. MegaModel: ϵ

This association denotes $\in$ in the set theory. Nothing less, nothing more. The relationship between the megamodel, the set theory and the language theory is described in [8]. A language is a set and "*hooo*" ϵ *hoL* holds. As another example, the Java language is the (infinite) set of all java programs. UML is a modelling language. It is the set of all UML models, so *Figure* 2 ϵ UML.

2.4 Metamodels and ConformsTo (χ)

Languages are very abstract systems. One need practical means to deal with languages, leading to the (derived) notion of *model of language*. There is nothing new here because languages are just particular systems. For instance, the regular expression $h(o)*$ is a model of the language *hoL* described above, so $h(o)*$ μ *hoL*. A grammar is a model of a language, not a language. An XML DTD, lets say *x.dtd*, is a model of a language, not a language. It is well known that a given language can be modelled by many models (expressed themselves in the same language or using different languages). In the Grammarware technological space [12], this fact is expressed by saying that they are many grammars for a single language (and various grammar languages to express these grammars, such as BNF and YACC). As an example we also have $h(o)*$ $(o)*$ μ *hoL*. Instead of using regular expressions, the *hoL* language can be also modelled by a grammar expressed in BNF or using YACC.

Models of languages ($\mu\chi$) must not be confused with languages of models ($\chi\mu$). *Modelling languages* is the common name used for "languages of models". But in fact, the important concept in MDE is the concept of *models of languages of models* ($\mu\chi\mu$), that is, models of modelling languages. These models are called *metamodels*. This concept leads to the association *ConformsTo* (χ) in the megamodel (Figure 5).

A model must *conform to* its metamodel. These relation has different incarnations depending on the Technological Space considered. For instance in the Grammarware TS, a phrase must conform to the grammar; in the XMLWare TS, an XML document must conform to a DTD; in the Dataware TS, the content of a database must conform to the schema of this database.

As shown in Episode II [7], this association was identified as a foundation of MDE in [2], but our contribution was to show that this is not indeed a basic association as previously thought. *ConformsTo* is on the contrary a

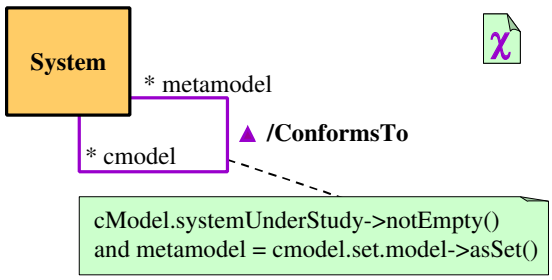


Fig. 5. MegaModel: χ

derived association as shown in Figure 2. In fact the *ConformsTo* takes its root in the set theory since it summarize a particular composition of μ and ϵ links. That is, it merges the notion of set and the notion of models [8]. The notion of metamodel given in this paper is indeed compatible with most of the definition found in the litterature. For instance the following definitions consistently express the fact that a metamodel is a model of a language of models: *"A meta-model is a model that defines the language for expressing a model" [16]. "A metamodel is a specification model for a class of SUS where each SUS in the class is itself a valid model expressed in a certain modelling language" [17].*

3 Transformations and IsTransformedIn (τ)

Though the initial version of the MDA standard did not put emphasis on transformations, this is really a core concept of MDE, just as model and metamodel. Furthermore, transformations are fundamental to software evolution. In particular Model Driven Software Evolution could also be called Transformation Driven Evolution. Unfortunately though notion of transformation is rather intuitive, there is no consensual terminology and this term is often used to refer to distinct concepts [8]. So let us clarify this notion. It is important to distinguish transformation instances, transformation (functions), transformation models, transformation modelling languages, and transformation metamodels. The goal of this paper is by no means to define a standard terminology. It is just to show that there is a serious issue here and that the megamodel can improve understanding and reasoning about MDE.

3.1 Transformation instances and τ

Following the set theory style, we can say that a system is transformed into another system by modelling this as a simple pair. For that it is enough to introduce the IsTransformedIn (τ) association in the megamodel. This is done on the left of the following figure.

As an example $1\ \tau\ 2$ means the integer 1 IsTransformedIn 2. The pair $(1, 2)$ will be called a *transformation instance*, or transformation application. 1 plays the role of *source* for this transformation while 2 plays the role *target*. If $p1$

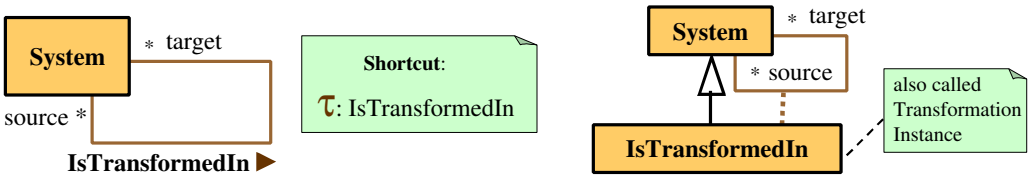


Fig. 6. MegaModel: Transformation instances (τ)

and $p2$ are programs, $p1\ \tau\ p2$ means that the program $p1$ IsTransformedIn the program $p2$. Then $(p1, p2)$ is a program transformation instance. It is important to note that they are no constraint on the transformation instances, and that no reference is made to the complexity of the transformation. $(p1, p2)$ could simply correspond to the addition of a white space, the modification of an algorithm, the renaming of a procedure, or it could be a complete reimplementation of the program.

As the reader might have noticed, the fact that τ is defined on system makes this association very general. It can be combined arbitrarily with the other associations introduced so far. For instance we can say that a model is transformed into another model, that a metamodel is transformed into another metamodel, that a model is transformed into a metamodel, etc. While this last example might seems strange at the first sight, just consider that some tools are able to extract a DTD (e.g. $X.dtd$) from an XML file $x1.xml$. The file $X.dtd$ is the incarnation of a metamodel in the XMLWare Technological Space, while $x1.xml$ is a model. This situation can be modelled by the following facts: $x1.xml\ \tau\ X.dtd$ and $x1.xml\ \chi\ X.dtd$. As it will be shown by a graph pattern in Figure 8, such transformation is the incarnation of "metamodel inference". Some commercial tools do that.

3.2 Transformation instances as systems

Considering transformation instances as first-class entities, bring even more power to the megamodel. Transformation instances are systems themselves. The associative class on the right of Figure 6 is both a class and an association. The class should be called *TransformationInstance* but unfortunately the association already received the IsTransformedIn name, and only one name can be defined in UML for a given element. Anyway, objects of this (associative) class can be the origin of destination of any link according to UML semantics. This is exactly what is needed. In particular τ links can be combined in many ways leading to complex τ-graphs. For instance the source and/or the target of a transformation instance could be another transformation instance. The power of higher order functions and currification is well known in Computer Science, and this is exactly what happen in the various technological spaces, though these terms are not necessarily used.

Seeing transformation instances as systems also means that τ can be combined with all other associations from the megamodel (e.g. ϵ, χ, μ). This

is necessary to model the realm of software development. For instance let us assume that we want to analyse the transformation instance from the program $p1$ to the program $p2$, that is the pair $(p1, p2)$. The result of this analysis might be an XML file $diff12.xml$ which models the differences between the source and the target. We want something smarter that the output of the unix "diff" tool. We have $diff12.xml\ \mu\ (p1, p2)$ and $p1\ \tau\ p2$. So $diff12.xml$ is "model of a transformation instance". Continuing with the same example, this XML file might be composed by other transformation instances. For instance the function $f1$ which is a part of $p1$, might have been transformed in the function $f2$ in $p2$. So we have $diff12.xml\ \delta\ (f1, f2)$, $p1\ \delta\ f1$, $p2\ \delta\ f2$ and $f1\ \tau\ f2$. Analysing the transformation instance $(p1, p2)$ and producing the summary $diff12.xml$ can be useful to understand program evolution. This can be modelled by the following facts $(p1, p2)\ \tau\ diff12.xml$ and $diff12.xml\ \mu\ (p1, p2)$. From the occurrence of pattern involving τ and μ in the opposite direction, it can be deduced that this is a "reverse engineering transformation", and since it applies to a transformation instance this concrete operation is an example of "transformation instance reverse engineering".

3.3 Transformation (functions)

So far, we have seen that individual systems can be transformed into other individual systems. Software evolution can be seen as a succession of transformation instances, but this is a very weak result which brings no concrete benefits. On the contrary, the challenge of MDE is to automate transformations as far as possible. This could be done only if transformation instances are considered in isolation. They should be described at a higher level of abstraction; not on individuals systems, but on set of systems.

A *transformation function*, or *transformation* for short, is a function in the mathematical sense of the term, that is a set of pairs with the constraint that a value map in the domain maps to at most one value in the range [18]. To be more precise *a transformation (function) is a set of transformation instances*. A transformation instance is an ElementOf (ϵ) zero or more transformations. The *domain* of a transformation (function) is the set of systems that can be transformed. The *range* of a transformation (function) is the set of systems that can be obtained via this transformation. This modelling directly comes from the set theory and the Z mathematical language [18]. We just use here the term transformation function or simply transformation instead of function because the term "software evolution through functions" is less popular that "software evolution through transformations". A transformation *is* however a function. Hence the fact that the correct term is "transformation function".

For instance $(1, 2) \in \tau$ means that the integer value 1 is transformed in 2. The set $\{(0, 1), (1, 2), (2, 3), (3, 4), ...\}$ is a transformation (function) if all its elements are transformation instances. Lets call this transformation $add1$ while $mul2$ will refer to the transformation $\{(0, 0), (1, 2), (2, 4), (3, 6), ...\}$. In

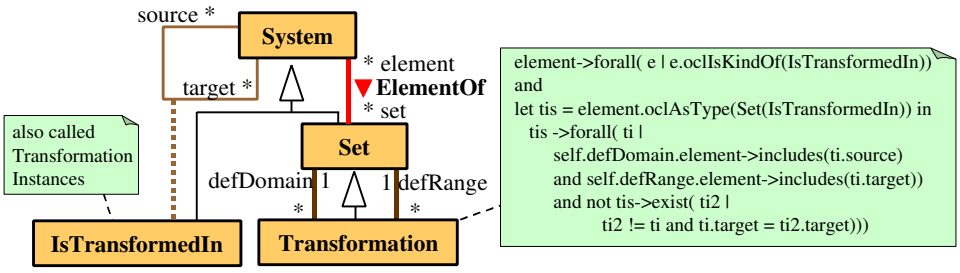


Fig. 7. MegaModel: Tranformations Functions vs. Transformation Instances

this example we can see that a transformation instance can be element of various transformation function since $(1,2)\ \epsilon\ add1$ and $(1,2)\ \epsilon\ mul2$. Classifying transformation instances in terms of existing transformation is a known problem in evolution analysis. For instance the goal might be to recognize refactorings from transformation instance. In this case one tries to figure out by observing the difference between two successive versions of a program, first what has changed, and then in which set this change (this transformation instance) can be classified. A refactoring such as "rename method" is a transformation function while the particular application of a refactoring is a transformation instance.

3.4 Transformation models

Transformation functions as defined above are abstract systems and there are therefore not operational. What is required in MDE, is a concrete means to describe transformations. This naturally leads to transformation models. *A transformation model is a model of a transformation (function).*

For instance lets call *dblC* the C program *int f(int x) {return x+x; }*, *dblP* the Pascal program $function\ f(x : integer) : integer\ begin\ return\ x * 2\ end$ and *dblC2* the C program *int f(int x){return x≪1;}*. It should be clear that *dblC* μ *mul2*, *dblP* μ *mul2* and *dblC2* μ *mul2*. As suggested by this example, they are plenty of ways and languages to write transformation models. This example is simple but it is not representative of software evolution through transformation. A compiler, a refactoring tool, or the YACC tool are better examples of transformation models for software evolution because they transform models.

Transformation models must not be confused with tranformation functions or with transformation instances. In the XMLWare Technological Space, an XSLT stylesheet is an example of a transformation model. It models a transformation (function) defined on XML files. This transformation could be be expressed in any other language, for instance XQuery. The application of the stylesheet on a particular XML file leads to a transformation instance. The application of a compiler on a particular program, or the application of YACC on a particular grammar are examples of transformation instances.

Every modification can be seen as a transformation instance. In fact the

huge majority of transformation instances applied during software evolution are ad-hoc. That is they are not elements of an existing transformation functions. Software engineers just change programs, without wondering if this is an instance of a transformation. Refactorings are examples of transformations, that can be modeled and therefore automated, but these are isolated examples. In fact currently software evolution is driven by ad-hoc transformation instances while the goal of Model Driven Engineering is to drive the process through a set of reusable transformation functions.

4 Applying the Megamodel to Software Evolution

We believe that all software evolution processes can be modelled as a graph using the megamodel presented above. Example of graphs are provided in [3] in the form of UML object diagrams, but Episode I and II present only a static vision. Before considering transformation and evolution, lets first consider such a static vision. Simply put each version of a large scale software is a complex system, so each version can be modelled as a graph built on the following elements:

- δ links, for instance to model the fact the software under study is made of packages, which are made of source files, which are made of functions, etc.
- μ links, for instance to model that a Z model is a specification of an Ada program.
- ϵ links, for instance to model that an XML model pertains to a Domain Specific Language (DSL). Languages should be considered as integral parts of software, especially in the long term since they will invariably evolve [5].
- χ links, for instance to model the fact that an XML model is conform to a DTD which model the DSL mentioned above. Or to model that the conformity of an Ada program is checked by the Ada compiler, which is a metaware tool [5].
- τ links, for instance to model the fact that a binary file has been produced from an Ada source file, itself produced from a Z specification model.

So each version can be represented by a graph. The evolution of the software can be modelled by a composition of these graphs using τ links. At the end, we just obtain a bigger graph with all kind of links. The megamodel is by no means restricted to model evolution or software evolution. Since everything is a system, everything could be transformed. That is, every system can be the source or the target of a τ link. This is required because in very large software companies everything evolve soon or later[5]. In fact evolution can be modelled by combination of τ links and other kind of links determining the kind of evolution. If we consider *evolution-in-the large* [5], languages evolve ($\epsilon\tau$). Metamodels evolve ($\mu\epsilon\mu\tau$). Transformation models evolve ($\tau\epsilon\mu\tau$). And so on.

Moreover when two systems connected by a link evolve, this leads to *co-evolution* issues. This is because consistency must be maintained between the ends of the link. Examples of co-evolution phenomenona, include for instance *model/code co-evolution* ($\tau\mu\tau$). *Metamodel/model co-evolution* [5] is another example ($\tau\mu\tau$).

The sequence of greek letters used here above are ambiguous, in particular because there is no formal rule for the ordering of letters. This is because the concepts described above corresponds to graph patterns, not simply sequences. We have identified a lot of interesting patterns that corresponds to known concepts. Some examples are provided in the next figure.

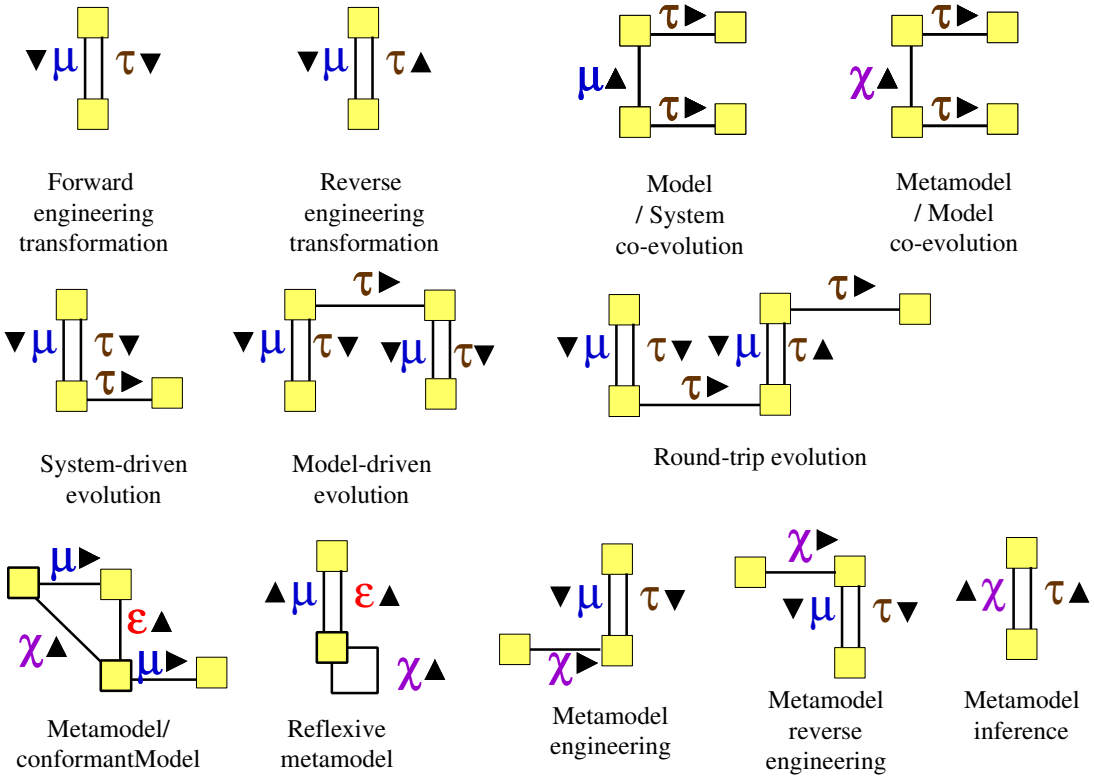


Fig. 8. MegaModel: Examples of interesting mega-patterns (τ)

5 Conclusion

In this paper we introduced a megamodel to describe MDE concepts and their relationships. This megamodel is summarized in Figure 9. The view presented here corresponds has been simplified for the purpose of this paper. A more complete view making explicit the relationships between the megamodel, the set theory and the language theory can be found in [8].

In fact, by using the megamodel we discovered that it was much more powerful than expected. It really helped us to connect concepts and technologies that were apparently disconnected. Surprisingly we discovered that a lot of

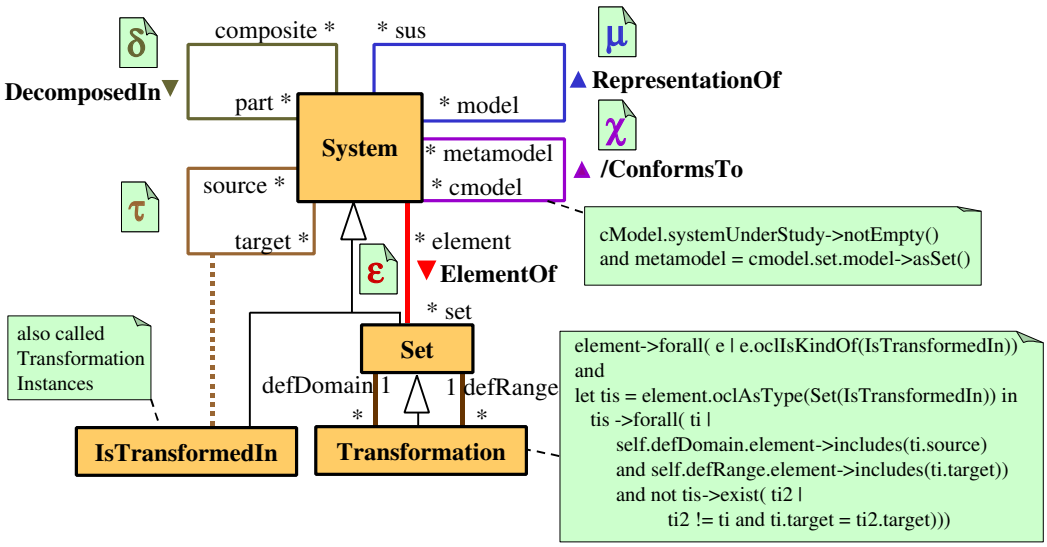


Fig. 9. MegaModel Overview

known issues could be model as graph patterns. And we are still discovering new ones. With respect to the form of the megamodel, the language used here is UML, but we are building other incarnations of the megamodel. At the time of writing this paper we are working on a version in Prolog, in Z [18], and in hieroglyphics [3][4]. The megamodel is expressed using different languages. Interestingly the Prolog megamodel is executable, so it can automatically recognize the patterns mentioned above and derive new facts from a given model expressed in the megamodel [8]. This program results from the transformation of the UML megamodel with the transformation of OCL constraints to prolog rules. In other words thats megamodel transformation, so we applied MDE to its own megamodel... In practice the problem is not really with the semantics of the megamodel, but with its interpretation [3]. That is formalizing the metamodel is not really an the important issue. MDE is not per se a formal system and the problem is much more about how to represent MDE real world, that about the language used to describe the resulting model. In other words the issue is how to extract a model from a software evolution process. To ease this task we are in the process of defining systematic mapping between the megamodel and its concrete incarnations in each technological spaces. Experiments we have done so far are very promising. This should not be surprising, because we are building the metamodels to reflect our practical knowledge about existing technological spaces.

We do not claim however that the megamodel presented here is complete or perfect. Like any other model, the megamodel certainly presents of lot of room for improvements. For instance, to model co-evolution phenomena it is necessary to include the notion of distance to express to what extent a model conforms to a metamodel (metamodel/model co-evolution), or a model is representation of code (model/code co-evolution). Adding the notion of metrics is also further work.

Finally the reader might still wonder what this research is all about. This is no code there. And nothing new either. Just consider things from a different perspective. The MDA standards is made of more than 2500 pages. It is full of complex technologies and they are plenty of commercial tools that claim to be MDA compliant. This standard is more than three years old. Major actors in software industry, such as Microsoft and IBM, announced MDE as being integral part of their strategy. Despite of that most people in academy still wonder what could be a metamodel transformation, a metamodel-driven evolution process or even a model-driven evolution process through transformation. While the term meta has been adopted by software industry leaders, it is still considered as suspicious by many. The goal of this paper is just to improve the understanding of MDE concepts and to make them more accessible. We believe that this is a strategical issue for software evolution. After about 50 years of empirical software evolution, it makes no doubt that software can be evolved in an ad-hoc way. What is needed, is a new paradigm for software evolution. MDE might be a candidate for that.

6 Acknowledgments

We would like to thanks the anonymous reviewers for their comments. We also would like to thanks Jean Bzivin, Jacky Estublier, German Vega, and all members of the AS MDA project, as well as attendee of the Dagsthul seminar 1401 on MDA for the fruitful discussions we had on this topic.

References

[1] C. Atkinson and T. Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, September 2003.

[2] J. Bézivin. In search of a basic principle for model-driven engineering. *Novatica Journal, Special Issue*, March-April 2004.

[3] Series From Ancient Egypt to Model Driven Engineering. www-adele.imag.fr/mda.

[4] J.F. Champollion. *Grammaire Egyptienne*. 1836. in French.

[5] J.M. Favre. Meta-models and models co-evolution in the 3d software space. In *Workshop on the Evolution of Large scale Industrial Software Applications, ELISA, Joint workshop with ICSM*, sept. 2003. available at www-adele.imag.fr/∼jmfavre.

[6] J.M. Favre. Foundations of Model (Driven) (Reverse) Engineering: Models - Episode I: Stories of the Fidus Papyrus and of the Solarus. In *Post-proceedings of Dagstuhl Seminar 04101*, 2004. Episode of [3].

[7] J.M. Favre. Foundations of the Meta-pyramids: Languages and Metamodels - Episode II: Story of Thotus the Baboon. In *Post-proceedings of Dagstuhl Seminar 04101*, 2004. Episode of [3].

[8] J.M. Favre. Towards a basic theory for modelling model driven engineering. In *Proceedings of WISME*, November 2004. available at www-adele.imag.fr/∼jmfavre.

[9] IEEE. Ieee recommended practice for architectural description of software-intensive systems, ieee std 1471, 2000.

[10] O Gerbé J. Bézivin. Towards a precise definition of the omg/mda framework. In *Proceedings of ASE01*, November 2001.

[11] A. Kleppe, S. Warmer, and W. Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[12] P. Klint, R. Lämmel, and C. Verhof. Towards an engineering discipline for grammarware. Technical report, submitted for publication, CWI. available at homepages.cwi.nl/∼ralf/.

[13] I. Kurtev, J. Bézivin, and M. Aksit. Technological spaces: an initial appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, Irvine, 2002.

[14] OMG. Mda web site. www.omg.org/mda.

[15] OMG. Omg, model driven architecture (mda). ormsc/2001-07-01, CWI, july 2001. available at www.omg.org/mda.

[16] OMG. Meta object facility (mof) specification, version 1.4. Technical report, april 2002. available at www.omg.org/mda.

[17] E. Seidewitz. What models mean. *IEEE Software*, September 2003.

[18] J.M. Spivey. *The Z notation: A Reference Manual*. Prentice Hall, 1992.

Modeling Software Evolution by Treating History as a First Class Entity

Stéphane Ducasse [1,4] Tudor Gîrba [2,4]

Software Composition Group
University of Bern, Switzerland

Jean-Marie Favre [3]

LSR-IMAG Laboratory
University of Grenoble, France

Abstract

The histories of software systems hold useful information when reasoning about the systems at hand or about general laws of software evolution. Yet, the approaches developed so far do not rely on an explicit meta-model and do not facilitate the comparison of different evolutions. We argue for the need to define history as a first class entity and propose a meta-model centered around the notion of history. We show the usefulness of our a meta-model by discussing the different analysis it enables.

Key words: software evolution, history meta-model

1 Introduction

The importance of observing and modeling software evolution started to be recognized in 1970's with the work of Lehman[15]. Since then more and more research has been spent to identifying the driving forces of software evolution, and to using this information to better understand software. However, the approaches developed so far, do not rely on an explicit meta model for evolution analysis and do not facilitate the comparison of different evolutions.

[1] Email: ducasse@iam.unibe.ch
[2] Email: girba@iam.unibe.ch
[3] Email: jean-marie.favre@imag.fr
[4] Ducasse and Gîrba gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02, Oct. 2002 - Sept. 2004) and "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006).

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The goal of this work is to propose a meta-model which allows for the usage of historical information just like any other kind of information.

Before going into details, we define three terms: *version*, *evolution* and *history*. A *version* is a snapshot of an entity at a particular moment in time. The *evolution* is the process that leads from one version to another. A *history* as the reification which encapsulates knowledge about evolution and version information. According to these definitions, we say that we use the history to understand the evolution (*i.e.,* history is a model of evolution).

This paper shows Hismo, a meta-model having in its center the notion of history, and argues that we need such a meta-model to reason about evolution of software systems. As a validation for our approach we present examples of historical measurements and history manipulations and show different usages for reverse engineering.

In the next section we enumerate the requirements a meta-model should support and we analyze existing techniques to analyze software evolution. In Section 3, we introduce Hismo, our history meta-model. In Section 4 we show examples of history measurements and in Section 5 we give examples of analyses enabled by our meta-model. In the end, we draw the conclusions and present the future work.

2 Software Evolution Analyses

Based on our analysis of the field, the requirements that a meta-model for analyzing the evolution of software systems fulfill are:s

- The meta-model should offer means to easily quantify and compare different property evolutions of different entities. For example, we must be able to compare the evolution of number of methods in different classes.
- The meta-model should allow for an analysis to be based on the evolution of different properties. Just like we can now reason about multiple structural properties, we want to be able to reason about how these properties have evolved. For example, when a class has only a few methods, but has a large number of lines of code, it should be refactored. In the same line, adding or removing the lines of code in a class while preserving the methods might be a sign of a bug-fix.
- The meta-model should provide change information at different level of abstraction such as packages, classes, methods (*i.e.,* not just text modifications).
- The meta-model should provide for the comparison in detail of two distinct versions of the same entity.
- The analysis should be applicable on any group of versions (*i.e.,* we should be able to select any portion of the history).

In the followings we enumerate different techniques used to analyze software evolution and how these techniques relate to the above requirements.

2.1 Evolution Chart Visualization

Since 1970 research is spent on building a theory of evolution by formulating laws based on empirical observations [15] [14]. The observations are based on interpreting evolution charts which represent some property on the vertical (*i.e.,* number of modules) and time on the horizontal (see Figure 1). Lately, the same approach has been employed to understand the evolution of open-source projects [2] [3]

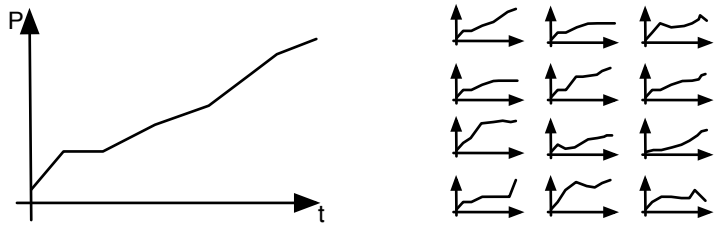


Fig. 1. Evolution chart example with some property on the vertical and time on the horizontal.

This approach is useful when we need to reason in terms of one property, but it makes it difficult to reason in terms of more properties at the same time, and provides only limited ways to compare evolutions of different properties. For example, it is suitable to use this technique to analyze the evolution number of modules in a system, but it is difficult to correlate the number of modules, with the total lines of code and with the number of developers.

In the left part of Figure 1 we display a graph with the evolution of a property P of an entity. From the figure we can draw the conclusion that P is growing in time. In the right part of the figure we displayed the evolution of property P in 12 entities. Almost all graphs show a growth of the P property but they do not have the same shape. Using the graphs alone it is difficult to say which are the differences and if they are important. Furthermore, if we want to correlate the evolution of property P with another property Q, then we have an even more difficult problem, and the evolution chart does not ease the task significantly.

2.2 Evolution Matrix Visualization

Visualization has been also used to reason about multiple evolution properties and to compare different evolutions of different entities. Lanza and Ducasse arranged the classes of the history of a system in an Evolution Matrix like in Figure 2 [12]. Each rectangle represents a version of a class and each line holds all the versions of that class. Furthermore, the size of the rectangle is given by different measurements applied on the class version. From the visualization

different evolution patterns can be detected: pulsar, idle, supernova or white dwarf.

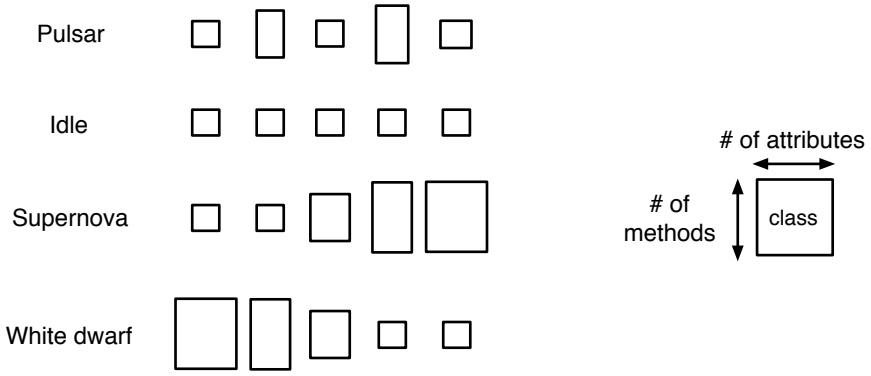


Fig. 2. Samples of class evolution patterns detectable in the Evolution Matrix.

With this visualization, we can reason in terms of two properties at the same time, and we can compare different evolutions. The drawback of the approach resides in the implicitness of the meta-model (*i.e.,* there is no explicit entity to which to assign the evolution properties) and because of that it is difficult to combine the evolution information with the version information. For example, we would like to know if the pulsar or idle classes are big or not.

Based on the detected patterns we can build a vocabulary for characterizing classes. Thus, in a system we can have pulsar classes or idle classes. But, pulsar and idle characterize a complete line and not just a cell in the matrix. Therefore, pulsar and idle characterize the way a class evolved over time and not a class. Based on this observation we concluded that we need a noun to which to assign the pulsar-like properties: the history.

Other visualizations approaches are based on similar meta-models. Jazayeri analyzes the stability of the architecture [11] by using colors to depict the changes. Taylor and Munro [18] visualizes version data with a technique called *revision towers*. Ball and Eick [1] develope visualizations for showing changes that appear in the source code. Collberg *et al.* use graph-based visualizations to display the changes authors make to class hierarchies [4]. Rysselberghe and Demeyer use a simple visualization based on information in version control systems to provide an overview of the evolution of systems [19].

2.3 Release History Meta-Model

Fischer *et al.* modeled bug reports in relation with version control system (CVS) items [7]. In Figure 3 we present an excerpt of the Release History Meta-model. The purpose of this meta-model is to provide a link between the versioning system to the bug reports.

This meta-model recognizes the notion of the history (*i.e.,* CVSItem) which contains multiple versions (*i.e.,* CVSItemLog). The CVSItemLog is related to a description and to BugReports. The authors used this meta-model to

recover features based on the bug reports [6]. These features get associated with a CVSItem.

A similar meta-model have been used to detect logical coupling between parts of the system [8]. The authors used the CVSItemLogs to detect the parts of the system which change together and then they used this information to define a coupling measurement.

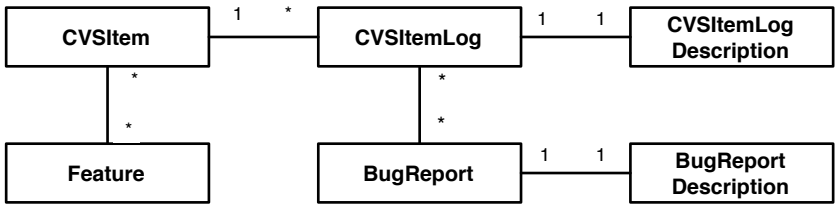


Fig. 3. Excerpt from the Release History Model-model.

The main drawback of this meta-model is that it does not take into consideration the structure of the software at the version level – *i.e.,* the system is represented with only files and folders, but no semantical units are represented (*e.g.,* classes or methods). Therefore, this meta-model does not offer support for different semantics of change – *i.e.,* it gives no information about what exactly changed in a system.

Zimmerman *et al.* aimed to provide mechanism to warn developers that: "Programmers who changed these functions also changed ...". The authors placed their analysis at the level of entities in the meta-model (*e.g.,* methods) [20]. Unfortunately, they did not explicitly describe their underlying meta-model.

3 Hismo - History Meta-Model

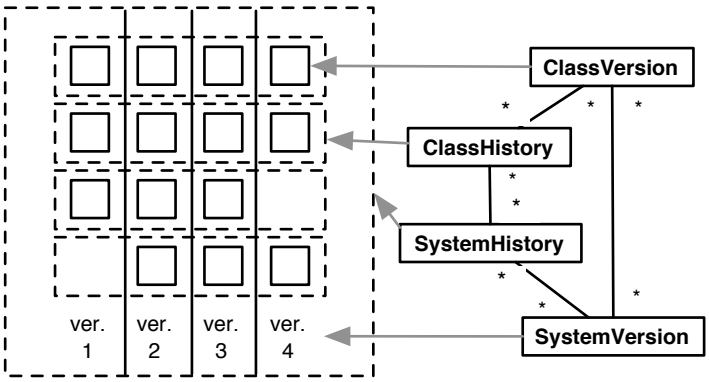


Fig. 4. History and the Evolution Matrix.

Figure 4 shows how a meta-model centered around the notion of history can be built: each cell in the matrix is a Class Version which makes for each line to represent a Class History. Moreover, the whole matrix is actually a line

formed by SystemVersions, which means that the whole matrix can be seen as a SystemHistory. In the right side of the figure we built a small meta-model which shows that a SystemHistory has more ClassHistories.

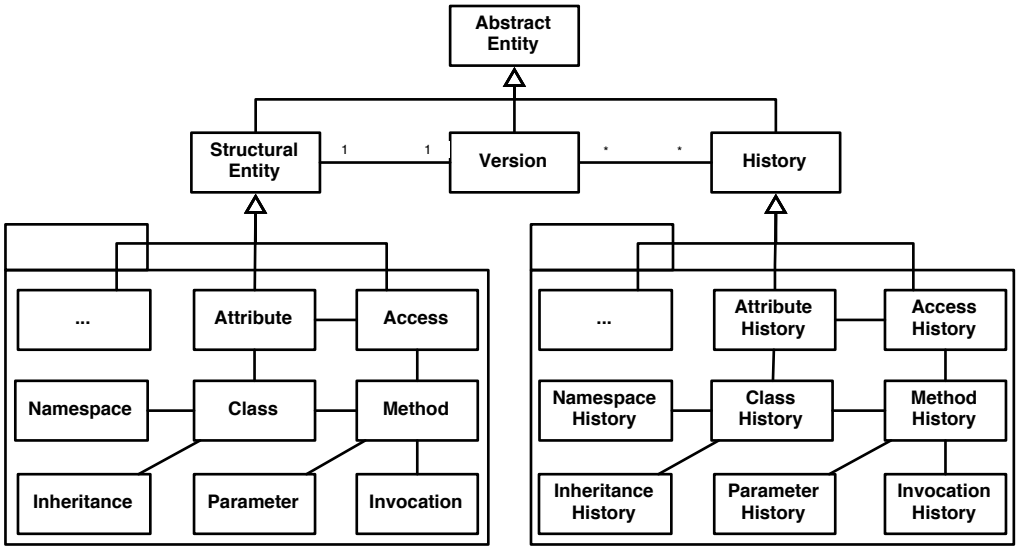


Fig. 5. An excerpt of Hismo and its relation with a source code meta-model. We did not represent all the inheritance relationships to not affect the readability of the picture.

In Figure 5 we show a reduced history meta-model based on a source-code meta-model. In our case we used FAMIX [5]. Each version entity has a correspondent history entity. Also, the relationship at version level (*e.g.*, a Class has more Methods) has a correspondent at the history level (*e.g.*, a ClassHistory has more MethodHistories).

A history does not have direct relation with a version entity, but through a Version wrapper. In Figure 6 we show the details of the relationship between History and Version.

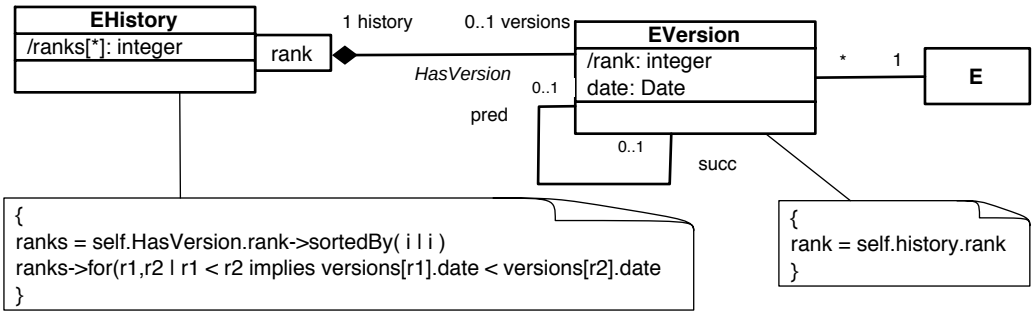


Fig. 6. Details of the relationship between the History, the Version and the structural entity (E). We used OCL notation.

4 History Measurements in Hismo

In this section we show some examples of how we use our meta-model to measure the evolution. We also show how the meta-model supports history selection and how measurements can be applied on any such selection.

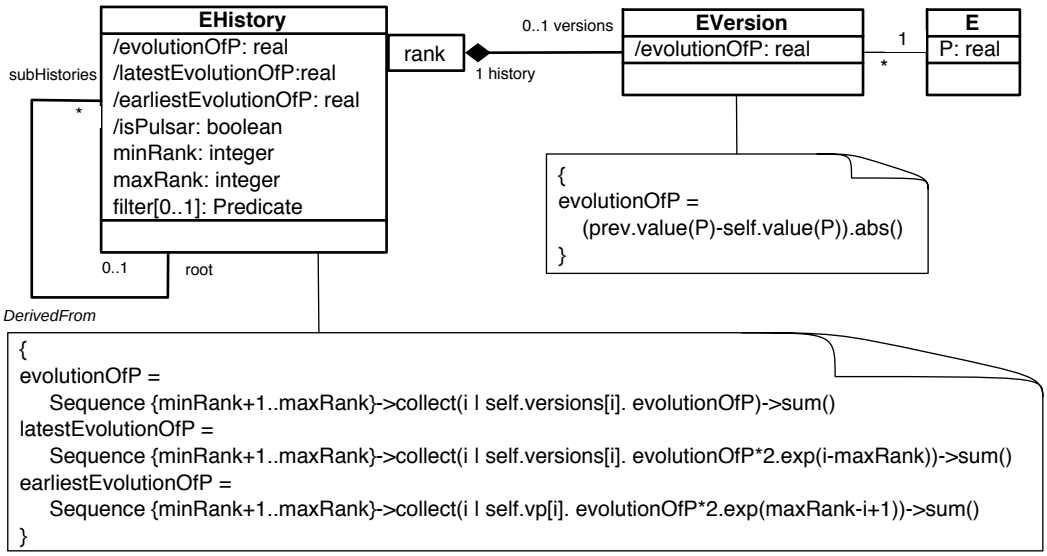


Fig. 7. Examples of history measurements definitions.

In Figure 7 we introduce three measurements: Evolution of P, Latest Evolution of P and Earliest Evolution of P.

Evolution of a property P (EP) – this measurement is defined as the sum of the absolute difference of P in subsequent versions. This measurement can be used as an overall indicator of change.

Latest Evolution of P (LEP) – while EP treats each change the same, with LEP we focus on the latest changes by weighting function ($2^{i-maxRank}$) which decreases the importance of a change as the version (i) in which it occurs is more distant from the latest considered version ($maxRank$).

Earliest Evolution of P (EEP) – it is similar to LEP, only that it emphasizes the early changes.

Figure 7 also shows that given a history we can filter it to obtain a sub history. As the defined measurements are applicable on a history, and a selection of a history is another history, the measurements can be applied on any selection too.

In Figure 8 we show an example of applying the defined history measurements to 5 histories of 5 versions each.

- During the displayed history of D (5 versions) P remained 2. That is the reason why all three history measurements were 0.
- Throughout the histories of class A, of class B and of class E the P property

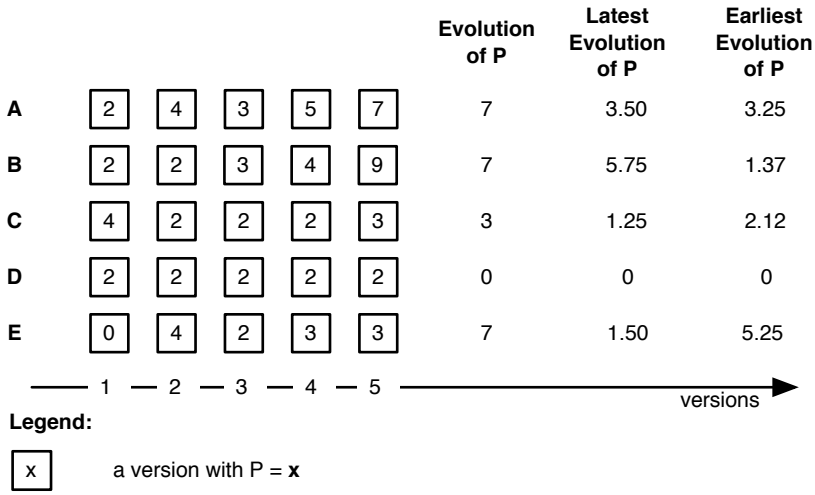


Fig. 8. Example of history measurements.

was changed the same as shown by the Evolution of P (EP). The Latest and the Earliest Evolution of P (LEP and EEP) values differ for the three class histories which means that (i) the changes are more recent in the history of class B (ii) the changes happened in the past in the history of class E and (iii) in the history of class A the changes were scattered through the history more evenly.

- The histories of class C and E have almost the same LEP value, because of the similar amount of changes in their recent history. The EP values differ heavily because class E was changed more throughout its history than class C.

The P property can be a property like: number of methods of a class, complexity of a method etc. Furthermore, we can define other measurements like: addition/removals of P, stability/instability of P etc.

5 Hismo Applications

The benefit of the historical measurements is that we can understand what happened with an entity *without* a detailed look at each version – *i.e.,* the measurements summarize time into numbers which are assigned to the corresponding histories. In the rest of the section, we describe three applications based on our meta-model:

- Build more complex historical measurements,
- Visualize different historical measurements to determine correlations and patterns of evolution.
- Build automatic queries which combine different evolution characteristics with version information to improve the detection of design flaws.

5.1 Yesterday's Weather

The above mentioned measurements were used to define another measurement: Yesterday's Weather (YW) [10]. YW is defined to be the retrospective empirical observation of the phenomenon that at least one of the classes which were heavily changed in the recent history is also among the most changed classes in the near future.

The approach consists in identifying, for each version of a subject system, the classes that were changed the most in the recent history and in checking if these are also among the most changed classes in the successive versions. The YW value is given by the number of versions in which this assumption holds divided by the total number of analyzed versions. If YW raises a high value, we say it is useful to start reengineering from the classes which changed the most in the recent past, because there is a high chance that they will also be among the most changed in the near future.

YW is a historical measurement obtained by combining different historical measurement which are applied on sub histories.

5.2 Hierarchy Evolution Complexity View

Based on Hismo, a visualization has been proposed to detect patterns of hierarchy evolution [9]. The visualization is based on the polymetric view [13]. Figure 9 shows the visualization applied on the history of six class hierarchies. The nodes represent class histories and the edges inheritance histories. Both the nodes and the edges are annotated with historical measurements. The visualization combines the evolution of different properties for building a vocabulary to characterize the evolution of class hierarchies: old hierarchies, stable hierarchies etc.

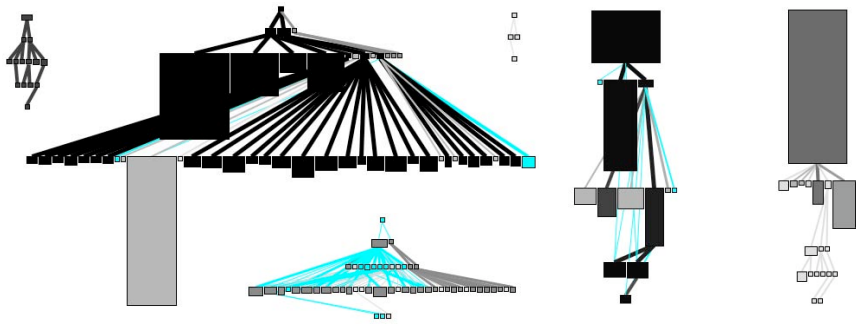


Fig. 9. Examples of class hierarchies evolution. Nodes represent class histories and the edges represent inheritance histories. Node width = Evolution of Number of Methods; Node height = Evolution of Number of Statements; Node color = Class Age; Edge width = Age; Edge color = Age.

Another usage of history measurements was proposed for improving design flaws detection [17]. In particular, the work shows how the detection of DataClasses and GodClasses [16] based on version measurements can be improved by taking into account information like: stability or the persistence of the flaw.

For example, Marinescu defines GodClasses as "those classes that tend to centralize the intelligence of the system." [16]. He also defined measurements-based expressions to detect GodClasses. We used the historical information to qualify GodClasses as being harmless if they were stable for a large part of their history, because that means those classes were not a maintainability problem in the past (*e.g.,* 95%). Below we present the expression we used.

```
context ClassHistory
    derive isHarmlessGodClass: (self.versions->last().isGodClass) &
                               (self.stabilityOfNOM > 0.95)
```

In this expression, we show how we can combine the historical information with other kinds of information to build our reasoning.

6 Conclusions and Future Work

Understanding software evolution is important as evolution holds information that can be used either in reverse engineering or in developing laws of evolution.

We browsed various techniques that have been used to understand the evolution, we discussed their shortcomings and we gathered requirements for our meta-model:

- Comparison of different evolutions of the same property,
- Combination of different property evolutions,
- History navigation/selection,
- Different semantics of change,
- Detailed version comparison.

Based on these requirements we proposed Hismo, a meta-model centered around the notion of history, and we gave examples of measurements applied on history. As a validation we showed the usages of our meta-model in different analyses.

In Figure 10 we show how a GeneralizedHistory is not just a sequence, but a graph; thus we can model branches. In the future, we would also like to explore the information given by branches.

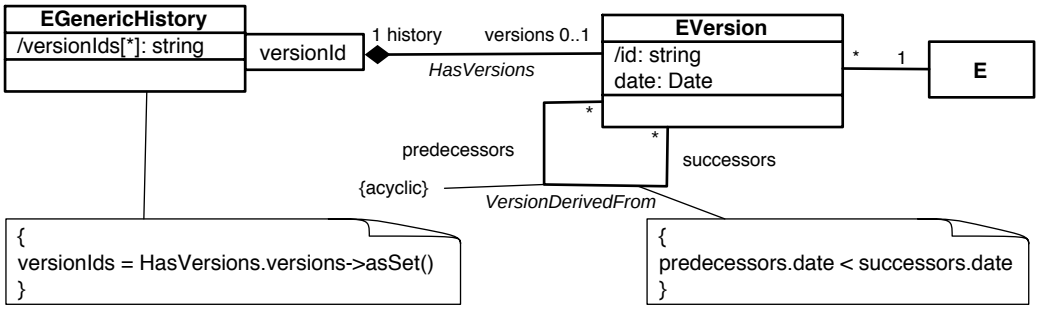


Fig. 10. Generalized Hismo.

References

[1] T. Ball and S. Eick. Software visualization in the large. *IEEE Computer*, pages 33–43, 1996.

[2] Andrea Capiluppi. Models for the evolution of os projects. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 65–74, 2003.

[3] Andrea Capiluppi, P. Lago, and M. Morisio. Evolution of understandability in oss projects. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 58–66, 2004.

[4] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 77–86. ACM Press, 2003.

[5] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — the FAMOOS information exchange model. Technical report, University of Bern, 2001.

[6] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 90–99, November 2003.

[7] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 23–32, September 2003.

[8] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance 1998 (ICSM '98)*, pages 190–198, 1998.

[9] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of ICSM 2004 (International Conference on Software Maintenance)*, 2004.

[10] Tudor Gîrba and Michele Lanza. Visualizing and characterizing the evolution of class hierarchies. In *Fifth International Workshop on Object-Oriented Reengineering (WOOR 2004)*, 2004.

[11] Mehdi Jazayeri. On architectural stability and evolution. In *Reliable Software Technlogies-Ada-Europe 2002*, pages 13–23. Springer Verlag, 2002.

[12] Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 2002 (Langages et Modèles à Objets*, pages 135–149, 2002.

[13] Michele Lanza and Stéphane Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003.

[14] Manny M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.

[15] Manny M. Lehman and Les Belady. *Program Evolution — Processes of Software Change*. London Academic Press, 1985.

[16] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. Ph.D. thesis, Department of Computer Science, "Politehnica" University of Timişoara, 2002.

[17] Daniel Raţiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using history information to improve design flaws detection. In *Proceedings of CSMR 2004 (European Conference on Software Maintenance and Reengineering)*, pages 223–232, 2004.

[18] Christopher M. B. Taylor and Malcolm Munro. Revision towers. In *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 43–50. IEEE Computer Society, 2002.

[19] Filip Van Rysselberghe and Serge Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings of The 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, 2004. to appear.

[20] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, 2004.

Towards the Integration of Versioning Systems, Bug Reports and Source Code Meta-Models

Giuliano Antoniol [1] Massimiliano Di Penta [2] Harald Gall [3]
Martin Pinzger [4]

[1,2] *RCOST - Research Centre on Software Technology*
University of Sannio, Department of Engineering
Palazzo ex Poste, Via Traiano 82100 Benevento, Italy

[3] *University of Zurich*
Department of Informatics

[4] *Technical University of Vienna*
Information Systems Institute

Abstract

Versioning system repositories and bug tracking systems are valuable sources of information to study the evolution of large open source software systems. However, being conceived for specific purposes, i.e., to support the development or trigger maintenance activities, they do neither allow an easy information browsing nor support the study of software evolution. For example, queries such as locating and browsing the faultiest methods are not provided.

This paper addresses such issues and proposes an approach and a framework to consistently merge information extracted from source code, versioning repositories and bug reports. Our information representation exploits the property concepts of the FAMIX information exchange meta-model, allowing to represent, browse, and query, at different level of abstractions, the concept of interest. This allows the user to navigate back and forth from versioning system modification reports to bug reports and to source code. This paper presents the analysis framework and approaches to populate it, tools developed and under development for it, as well as lessons learned while analyzing several releases of Mozilla.

Key words: Source Code Analysis, Release History, Bug Reports, Object-Oriented Meta-Models

[1] Web: http://alpha.rcost.unisannio.it/antoniol
[2] Web: http://www.rcost.unisannio.it/mdipenta
[3] Web: http://www.ifi.unizh.ch/swe/
[4] Web: http://www.infosys.tuwien.ac.at/Cafe/

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

The use of configuration management tools in software development and maintenance activities constitutes a consolidated practice, also supported by software engineering principles. Versioning systems such as the Concurrent Versioning System (CVS) are commonly adopted, especially for large-scale projects. Often versioning systems are complemented by bug reporting tools, that constitute an essential support for corrective maintenance.

These two families of tools are valuable sources of information to study software evolution; CVS can be exploited to get insights about evolution in terms of size, complexity, amount of changes, and whatever can be mined from source code or from change logs. On the other hand, bug reporting systems provide insights on reliability, as well as information on how an organization manages defects (e.g., what is the average defect fixing rate, statistics about defect severity).

There are several freely available or commercial tools, for example *Bugzilla* [24] or *Rational ClearCase* [5], integrating CVS repositories and bug tracking systems. However, what would be interesting is to also integrate meta-models describing facts extracted from source code. This would allow maintainers and project managers to get an overall view. For instance, it would be feasible to identify a subset of components (classes, functions or methods above a given size) which exhibit more defects than others. Similarly, one could analyze the relationship between defects and some language constructs e.g., the use of pointers or inheritance versus defect proneness.

It is the opinion of the authors that the task of integrating heterogeneous sources of information may be simplified by the adoption of a meta-model allowing to accommodate source code abstractions, source code changes as well as details on bug reports. Different schemata and meta-models have been proposed to represent both procedural and object-oriented (OO) software; however, the proposed schema either have been tailored to a specific language such as C, C++ [8], Smalltalk or Java, or have not been annotated with source code level details or have not been integrated with other relevant source of information such as bug reports and CVS data.

This paper proposes to enrich the FAMIX information exchange meta-model [21] with detailed information extracted from a combination of heterogeneous sources, namely source code, CVS and bug tracking repositories. FAMIX provides the concept of property which naturally leads to decorate entities of interest (e.g., classes or methods) with a variety of details. Clearly, entity decoration depends on the particular entity. Some decorations, such as file name and subsystem name, are common to all entities. Changes and defects are decorations applicable to class, method, and function entities. Finally, other properties, such as the number of parameters, belong exclusively to templates, methods and functions.

To reason about changes, bugs and source code entities, to relate facts or extract

[5] `http://www.rational.com`

statistics, we must build a traceability map between different concepts. At this purpose, we follow an approach inspired by the `diff` and `patch` Unix utilities. Changes are identified by means of file name and line numbers, hereby referred as *location by site*. This permits the integration of the information extracted from source code, bug reports and CVS change logs. The above choice stems from the following observation. Modification Reports (MR) often detail involved files and changed lines of code; once files and changed lines have been identified, the changed context (class and method) can be located by parsing sources extracted from a CVS repository. Vice-versa, given a contextual information (file, beginning and ending line) it is possible to identify the Problem Report (PR) impacting that code region.

To verify feasibility, discover particular strengths, problems and pitfalls, the framework was applied to several releases of Mozilla. Mozilla is a multi-million LOCs open source project [6] managed via a CVS repository and a bug tracking system. Source code information, CVS information and bug information were used to decorate our instantiated meta-model. Particularly, releases from 1.0 to 1.3.1 were used as case study. At the time of writing, integration has been achieved at the file level; more fine grained integration, namely at the class, method or line level is only partially supported.

The remainder of the paper is organized as follows. After a review of the related work, Section 3 proposes our approach and presents the framework. Section 4 describes the Mozilla case study. Section 5 describes our tools developed to extract information and to populate the repository; Section 6 reports on and summarizes the experiences gained in populating the first release of the repository. Finally Section 7 concludes and outlines foreseeable research activities.

2 Related Work

The present work stems from the release history database of Fisher et al. [10]. The authors proposed to combine CVS revision data with bug reporting data and to add some missing information such as, for example, merge points. The same authors also performed, on the same data, an analysis devoted to track features [9]. Finally, Gall et al. [14] analyzed CVS release history data for detecting logical coupling.

In [18] maintenance requests were classified according to maintenance categories as IEEE Std. 1219 [1], then rated on a fault-severity scale using a frequency-based method. Ball et al. [5] proposed an approach for visualization of data extracted from a version control system, while a three-dimensional color visualization of release history was proposed in [15].

Some other relevant studies have been performed in the past with the purpose of understanding the architecture and the evolution of the Mozilla open source project. In particular, Godfrey et al. integrated different reverse engineering tools and used them to extract Mozilla's architecture [16]. Mockus et al. studied the evolution of

[6] `http://www.mozilla.org`

two large open source projects, namely Apache and Mozilla [17]. Eick et al. [7] performed a graphical analysis of computer log.

3 The Framework

The main idea adopted to locate, browse, and integrate heterogeneous information is to rely on *location by site*: classes, methods, functions as well as defects and changes must be located by file and line number. The following subsections provide details on the approaches, implemented or under development, adopted to build traceability mapping between code area regions (classes, methods, functions), PRs, and MRs. At the time of writing sources code, PRs, and MRs are managed at different granularity levels. Integration at file level has already been achieved; browsing at class, method or line of code level is only supported from source code entity to code region.

3.1 Bugzilla and CVS repositories

Release history data is retrieved from versioning systems such as the CVS [12] and bug tracking systems such as Bugzilla [24]. In particular, we obtain MRs from CVS and PRs from Bugzilla. Fig. 1 depicts the core of our *Release History Database* with linked MRs and PRs. MRs are stored in the *cvsitemlog* entity and PRs in the *bugreport* entity of our RHDB. Information about the file to which a MR belongs is stored in *cvsitem*.

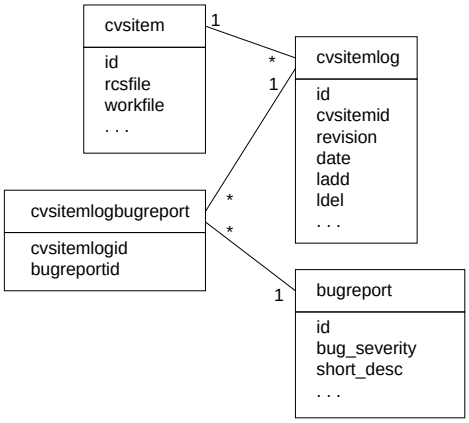


Fig. 1. Core of the RHDB

Links between MRs and PRs are stored to the table *cvsitemlogbugreport*. Establishing such links is an important issue of the RHDB population process. Concerning CVS and Bugzilla, this needs to be done separately. A link is stored whenever a reference to a PR is found in a MR. PR figures in MRs are searched using regular expressions (e.g., #128764). Because these numbers are entered as free text, results contain correct and false positive matches as well. To improve data quality, all matched numbers are validated using information available with PRs such as

patches that contain the names of files they are applied to. If this file name corresponds to the name of the file of the MR the link is validated [9]. More details on bug report and CVS processing can be found in [9,10,14].

Summarizing, the RHDB contains versioning, change and defect relevant data about each file of each release that has to be integrated with source model data as described next.

3.2 Source Code Modeling

All the concepts available at design level, such as those modeled via UML diagrams, function invocation, and software metrics are extracted and represented. The source code meta-model was inspired by the FAMIX [21] information exchange meta-model [7].

FAMIX prescribes CASE Data Interchange Format (CDIF) [6] as the basis for information exchange. Other standards and interchange formats exist, for example XMI, an XML based interchange format [20], or the Rigi Standard Format (RSF) [22]. The RSF origins from the Rigi program visualization, reverse engineering and program understanding environment, and is a triple based specification language that can be easily customized and imported into different tools. Fig. 2 shows an RSF excerpt of a class representation. We use RSF to represent the concepts of interest such as classes, class attributes and methods, types, or different kinds of software metrics, etc.

```
type nsAutoRefCnt "Class"
contain ./dist/include/xpcom/nsISupportsImpl.h
                  nsAutoRefCnt
lineno  nsAutoRefCnt    "88"

type nsAutoRefCnt::mValue "Attribute"
belongsToClass  nsAutoRefCnt::mValue
         "nsAutoRefCnt"
type nsrefcnt "DataType"
hasType nsAutoRefCnt::mValue "nsrefcnt"
accessControlQualifier
         nsAutoRefCnt::mValue "PRIVATE"

type nsAutoRefCnt::nsAutoRefCnt() "Method"
isAbstract nsAutoRefCnt::nsAutoRefCnt() "TRUE"
belongsToClass  nsAutoRefCnt::nsAutoRefCnt()
         "nsAutoRefCnt"
type nsAutoRefCnt "DataType"
hasType nsAutoRefCnt::nsAutoRefCnt()
         "nsAutoRefCnt"
accessControlQualifier
         nsAutoRefCnt::nsAutoRefCnt() "PUBLIC"
```

Fig. 2. Excerpt of a class RSF representation

Fig. 3 shows, at a high level of abstraction, the steps carried out and the extracted information for the Mozilla browser. To avoid problem of missing files, wrong dependencies and compilation errors, the approach is a two phase approach. First, the source code undergoes a preliminary compilation to produce the target executables. Then source code is parsed and information extracted. The two phases

[7] http://www.iam.unibe.ch/∼famoos/

approach ensures that the application is properly configured for the current instance of architecture, operating system and, in general, hardware and software environment. Clearly, when PRs refer to configuration-dependent source code, consistency needs to be ensured between PRs and the source code facts. This paper focuses on source code (and thus PRs) related to a single configuration, for a Linux operating system and Intel architecture.

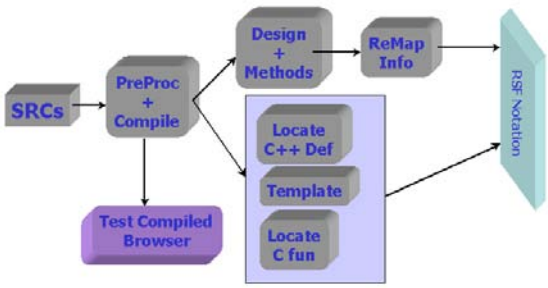


Fig. 3. C++ information extraction

To reuse already available tools, the extracted information is first represented via an extension of an intermediate language, name Abstract Object Language (AOL). AOL is a general-purpose design description language capable of expressing concepts available at the design stage of OO software. It has been extended to represent software metrics, structures, templates and other facts such as methods or functions calls. More details on AOL can be found in [2,3,4,11].

Fig. 4 details the steps encompassed by the box *PrepProc + Compile* of Fig. 3. The second compilation relies on wrappers wrapping C and C++ compilers. This is needed to avoid error prone activities required to modify by hand compilation scripts and makefiles. The preprocessed files contain both application and system information. Thus a further step may be needed to get rid of unusable information, i.e., to remove system include files.

Summarizing, RSF information comprises:

- a reverse engineered class model including inheritance, association and aggregation relationships;
- function and method level software metrics such as the number of passed parameters, the maximum nesting level, or the number of statements;
- details on template and structures; and
- location by site of classes, methods and functions.

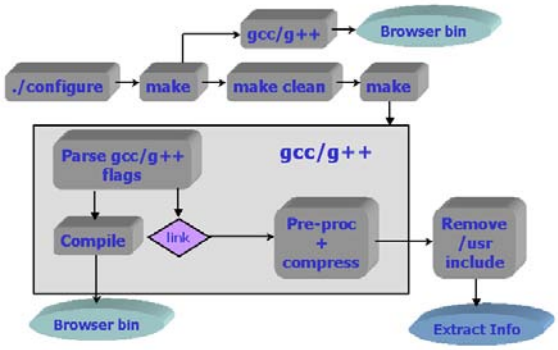


Fig. 4. C++ processing

3.3 Integrating the Information Sources

For the integration of source model and release history data, an entity common to both information spaces has to be determined. As already stated, in the context of this paper, we focus on source files to be the common entity because files are subject of versioning, change and defect data as retrieved from versioning and bug reporting systems. A finer-grained level of integration is under development and subject of ongoing work.

In our approach, source model information is available in FAMIX conform RSF files. Release history information is stored in a relational database and can be queried using SQL. The key connector that links both information sources is the unique name of files contained in both repositories. Based on these unique file names the data integration process is performed. The process is straightforward, and it consists of the two steps: 1) query RHDB and output results in RSF; 2) integrate results with source model RSF files.

In the first step we query the RHDB database with respect to the dependencies due to MR and PR data. The input to the query is a list of unique names of source files of interest. According to Fig. 1, we query the `cvsitem` table to get the file identifiers which on their own, are used to query the `cvsitemlog` joined with `cvsitemlogbugreport` and `bugreport` relations to get change related dependencies between selected source files.

Regarding both MR and PR, we introduce two new relationship types `rhdb-Coupled` and `rhdbDependent`. The fundamental principle of both relationship types is that from the point of view of changes two source files are *logically coupled* or *dependent* if they have been affected by the same source code modification [13]. Consequently, if two source files have been checked into the source repository (i.e., CVS) at about the same time, they are logically coupled. Furthermore, if two files are referenced by the same PR, then from the point of view of changes they are dependent. Additionally, we compute the number of affected MRs and PRs to determine the weight of these relationships. Results of the queries are output in the form of RSF tuples.

The second step of the integration process is concerned with integrating computed RSF tuples to the source model data. This includes determining affected source files in the source model database and computing the new edge identifiers for each integrated `rhdbCoupled` and `rhdbDependent` relationship.

The result of the integration process comprises a source model repository enriched with release history data that, on its own, comprises `rhdbCoupled` and `rhdbDependent` relationships between source file entities, as well as a `weight` attribute for each relationship.

4 The Case Study

To evaluate the feasibility of integrating source code level and quality related information, several versions of Mozilla, an open source web browser, were analyzed. The extracted key features are reported in Table 1. Mozilla was mostly developed in C++; the C code accounts only for a small fraction of the overall size. XML, HTML and scripting language configuration and support programs are also present. The latest Mozilla releases include more than 10,000 source files for a size up to 3.7 MLOC located in 2,500 subdirectories. Mozilla basically consists of 90 modules maintained by 50 different module owners. The Bugzilla bug tracking system contains more than 180,000 PRs and the CVS repository contains about 430,000 MRs.

Release	# C files	# h files	# C++ files	Size	Classes	Methods	Func.	Inheritances	Associations	Aggregations
1.0	1987	7,519	3,982	3.5	4,545	50,912	5,737	5,031	6,993	3,404
1.0.1	1995	7,603	4,022	3.5	4,561	54,742	5,740	5,051	7,006	3,440
1.0.2	1987	7,635	4,049	3.5	4,572	51,198	5,740	5,065	7,029	3,461
1.1	1997	7,674	4,054	3.6	4,594	52,453	5,742	5,095	7,048	3,466
1.2a	1984	7,769	4,058	3.6	4,475	51,104	5,741	4,992	7,107	3,514
1.2b	1991	7,972	4,122	3.7	4,512	53,697	5,794	5,029	7,141	4,804
1.2	1991	7,981	4,129	3.7	4,526	51,689	6,192	5,044	7,155	4,817
1.2.1	1991	7,981	4,129	3.7	4,524	52,953	5,794	5,044	7,156	4,817
1.3a	1823	7,880	4,145	3.6	4,574	51,827	5,809	5,081	7,157	6,090
1.3b	1830	7,924	4,164	3.6	4,589	51,580	5,836	5,101	7,339	6,200
1.3	1830	7,911	4,158	3.6	4,577	53,106	5,836	5,088	7,323	6,181
1.3.1	1830	7,935	4,198	3.7	4,577	51,453	5,836	5,088	7,323	6,181

Table 1
Mozilla key features

5 The Tools

Several tools were reused, modified or developed to extract and integrate information from the different sources.

5.1 Compiler Wrappers

C and C++ compiler wrappers, mimicking the same compiler interface, have been developed with Perl. `gcc` and `g++` specific options as well as linker options are fully supported and managed. Preprocessed source code is compressed to reduce disk space usage.

5.2 C++ Information Extraction

A tool inspired by island-driven parsing [19] has been reused and modified to reverse engineer a class diagram and extract class level and method level metrics. The island-parsing approach allowed to overcome most of the difficulties related to parsing C++ code (intrinsic language difficulties, dialects such as the GNU dialect encountered when parsing *Mozilla*, etc.). The tool was developed in previous projects to extract AOL. More details can be found in [2,3,4,11].

5.3 FAMIX Export and RSF Integration

The exporter and integration tool comprises two Perl scripts that process AOL files and output FAMIX data in RSF, as well as integrate the data of two RSF files into one. AOL is the format used by the C/C++ parser to output extracted facts. We used the extension points of the FAMIX model to specify additional attributes that hold the various metrics extracted for each source model entity. Furthermore, we also added two new relationships between source files to store logical couplings `rhdbCoupled` and hidden dependencies `rhdbDependend`.

Two steps are performed by the exporter, namely: 1) mapping AOL to RSF data; 2) integrating RSF files into one source model data file. Basically, the parser generates separate AOL files for storing extracted information about source files, classes, methods and functions. In the first step each of these files are input to the exporter that prints out plain RSF tuples of contained information. Preliminary checks are performed that consider the data stored in a files. For example class and inheritance relationships of the AOL class file are checked to existence of the base and subclass. AOL records that fail the check are not printed. In the second step the different RSF files are integrated into on file that contains the whole source model. During this integration process, existence checks on entities of relationships and attribute records are performed but in this step checks involve the whole data source.

Data about logical couplings and hidden dependencies between source files are also available in RSF format. Hence, the integration script is applied to add this relationships and the weight attribute to the existing source model data. The output is a RSF file that contains the integrated source model data in FAMIX conform RSF tuples which can be handled by existing visualization tools such as Rigi [23] or SHriMP [25].

6 Lesson Learned

This section summarizes lessons learned while integrating bug reports, CVS information and data from 12 Mozilla releases. Mozilla is a large software system, encompassing a variety of programming languages, styles and idioms. If a language contains a feature, someone will use it regardless of the impact it could have on understandability, portability maintainability, or evolvability. For example, C++ is a strongly typed, OO language. However, it retains C compatibility and considers a `struct` as a `class` only containing public attributes. This means that there may be classes inheriting from structs e.g., `struct nsBandData` and `class nsBlockBandData`. It should be noted that this is something different from wrapping a structure with a class, since it breaks information hiding and encapsulation. `nsBandData` is declared as a structure and there seems to be no reason why it should not be declared as a class but allowing access from C. In fact, a C++ compiler, compiles C code if it does not break C++ rules e.g., a `new` identifier causes compilation failure. However, C and C++ compilers have different conventions and thus the above practice may easy the task to break C++ encapsulation from C code.

Much in the same way, we observed structures containing method declarations. For example, `nsID` is a structure with three functions declared inside, one of which is both declared and defined inside `Equals`. Again, there is no obvious reason but allowing an easier access from C code.

Templates constitute a powerful mechanism to enable for parametric code development. However, authors believe that in certain cases it may also be abused. While templates should be used to create new *abstract data types*, weird uses of templates were found in Mozilla. For example, we found examples of templates, e.g., `nsCOMTypeInfo`, used to parameterize a structure.

These latter peculiar uses or *abuses* opened a discussion on what should be annotated on the meta-model, if and when the meta-model should be amended. The information is available; however it is not completely clear if and how certain facts have to be represented. For example, the `struct nsBandData` could be represented as a class and then flagged as an OO coding style violation or we have to extend FAMIX so that a class may inherits from structs. On the other hand, it should be noted that coding style violation are not usually part of source code meta-models. All in all, it further led to the need to modify the initial integration model for example to cope with classes derived from structures or structures with methods.

7 Conclusions and work-in-progress

Consistently integrating different repositories of large software systems, such as the open source software Mozilla and its CVS and Bugzilla data, is a challenging but fruitful task. It allows a user to represent, browse and query—at different levels of abstraction—the particular concept of interest, from the source code level to the bug report and modifications level. In this paper, we proposed a first step toward a

multi-level concept navigation framework that represents source entities in FAMIX meta-model compliant Rigi Standard Format (RSF).

We took the Mozilla browser as a case study and extracted its C++ sources over 12 releases into an RSF representation. This kind of data was combined with release history data populated from filtering related bug reports and modification reports into a release history database (RHDB). This release data integration venture allowed us to highlight problems encountered, difficulties and pitfalls.

Work–in–progress is devoted to manage the discovered information overflow problem, and complete the integration, ensuring a finer level of detail (i.e., tracing problems to classes and methods) and to ensure appropriate querying and browsing capabilities.

8 Acknowledgments

This research was partially supported by the RELEASE Excellence network founded by the European Science Foundation.

References

[1] "IEEE std 1219: Standard for Software maintenance," 1998.

[2] Antoniol, G., B. Caprile, A. Potrich and P. Tonella, *Design-code traceability for object oriented systems*, The Annals of Software Engineering **9** (2000), pp. 35–58.

[3] Antoniol, G., G. Casazza, M. Di Penta and R. Fiutem, *Object-oriented design patterns recovery*, Journal of Systems and Software **59** (2001), pp. 181–196.

[4] Antoniol, G., R. Fiutem and L. Cristoforetti, *Using metrics to identify design patterns in object-oriented software*, in: *Proceedings of 5th International Symposium on Software Metrics - METRICS98*, Bethesda MD, 1998, pp. 23–34.

[5] Ball, T., J. M. Kim, A. Porter and H. Siy, *If your version control could talk ...*, in: *ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*, Boston, MA, USA, 1997.

[6] Committee, C. T., "CDIF Framework for Modelling and Extensibility," Electronic Industries Association EIA/IS-107, 1994.

[7] Eick, S. G., M. C. Nelson and J. D. Schmidt, *Graphical analysis of computer log files*, Communications of the ACM **37** (1994), pp. 50–56.
URL citeseer.ist.psu.edu/eick94graphical.html

[8] Ferenc, R., S. E. Sim, R. C. Holt, R. Koschke and T. Gyimothy, *Towards a standard schema for c/c++*, in: *Working Conference on Reverse Engineering*, 2001, pp. 49–58.
URL citeseer.nj.nec.com/ferenc01towards.html

[9] Fischer, M., M. Pinzger and H. Gall, *Analyzing and Relating Bug Report Data for Feature Tracking*, in: *10th Working Conference on Reverse Engineering (WCRE), Victoria, Canada*, 2003, pp. 90–99.

[10] Fischer, M., M. Pinzger and H. Gall, *Populating a release history database from version control and bug tracking systems*, in: *Proceedings of IEEE International Conference on Software Maintenance*, Amsterdam, The Netherlands, 2003, pp. 23–32.

[11] Fiutem, R. and G. Antoniol, *Identifying design-code inconsistencies in object-oriented software: A case study*, in: *Proceedings of IEEE International Conference on Software Maintenance*, Bethesda MD, 1998, pp. 94–102.

[12] Free Software Foundation, "Version Management with CVS," 1.11.14 edition (2003), http://www.cvshome.org/docs/manual.

[13] Gall, H., K. Hajek and M. Jazayeri, *Detection of logical coupling based on product release history*, in: *Proceedings of the International Conference on Software Maintenance (ICSM '98)* (1998).

[14] Gall, H., M. Jazayeri and J. Krajewski, *Cvs release history data for detecting logical couplings*, in: *Proceedings of the International Workshop on Principles of Software Evolution*, Helsinki, Finand, 2003, pp. 13–23.

[15] Gall, H., M. Jazayeri and C. Riva, *Visualizing software release histories: The use of color and third dimension*, in: *Proceedings of IEEE International Conference on Software Maintenance*, Oxford, England, 1999, pp. 99–108.

[16] Godfrey, M. and E. Lee, *Secrets from the Monster: Extracting Mozilla's Software Architecture*, in: *Proceeding of Second Symposium on Constructing Software Engineering Tools*, 2000.

[17] Mockus, A., R. Fielding and J. Herbsleb, *Two case studies of open source software development: Apache and Mozilla*, ACM Transactions on Software Engineering and Methodology **11** (2002), pp. 309–346.

[18] Mockus, A. and L. Votta, *Identifying reasons for software changes using historic database*, in: *Proceedings of IEEE International Conference on Software Maintenance*, San Jose, California, 2000, pp. 120–130.

[19] Moonen, L., *Generating robust parsers using island grammars*, in: *Working Conference on Reverse Engineering*, 2001.

[20] OMG, "XML Metadata Interchange (XMI)," OMG Document ad/98-10-05, 1998.

[21] Software Composition Group, University of Berne, "The FAMIX 2.0 specification," 2.0 edition (1999), http://www.iam.unibe.ch/ scg/Archive/famoos/FAMIX/.

[22] Tilley, S. R., K. Wong, M.-A. D. Storey and H. A. Müller, *Programmable reverse engineering*, International Journal of Software Engineering and Knowledge Engineering **4** (1994), pp. 501–520.

[23] Wong, K., S. Tilley, H. A. Muller and M. D. Storey, *Structural redocumentation: A case study*, IEEE Software (1995), pp. 46–54.

[24] *Bugzilla Bug Tracking System*, http://www.bugzilla.org.

[25] *Shrimp views: Simple hierarchical multi-perspective*, http://shrimp.cs.uvic.ca/ (2004).

Behavioral Refinement of Graph Transformation-Based Models

Reiko Heckel [a] Sebastian Thöne [b]

[a] *Faculty of Comp. Science, Electrical Eng., and Math.,* reiko@upb.de

[b] *International Graduate School Dynamic Intelligent Systems,* seb@upb.de

University of Paderborn, Germany

Abstract

Model-driven software engineering requires the refinement of abstract models into more concrete, platform-specific ones. To create and verify such refinements, behavioral models capturing reconfiguration or communication scenarios are presented as instances of a *dynamic meta-model*, i.e., a typed graph transformation system specifying the concepts and basic operations scenarios may be composed of. Possible refinement relations between models can now be described based on the corresponding meta-models.

In contrast to previous approaches, refinement relations on graph transformation systems are not defined as fixed syntactic mappings between abstract transformation rules and, e.g., concrete rule expressions, but allow for a more loose, semantically defined relation between the transformation systems, resulting in a more flexible notion of refinement.

Key words: MDA and model transformation, consistency and co-evolution, refinement of graph transformation systems

1 Introduction

Model-driven software development is based on the idea of refining abstract models into more concrete ones, a recent example being the *Model-Driven Architecture (MDA)* put forward by the OMG [1] . Here, platform-specific details are initially ignored at the model level to allow for maximum portability. Then platform-independent models are refined by adding implementation details required for the mapping to a given target platform. Thus, at each level, more assumptions on the resources, constraints, and services of the chosen platform are incorporated into the model.

[1] www.omg.org/mda/

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The set of models conforming to a modeling language is often defined by a *meta-model*, i.e., a class diagram with constraints describing the individual elements of the model and their composition. For behavioral models, this approach is extended towards a *dynamic meta-model*, formalized as a *typed graph transformation system* [3]. Informally, a typed graph transformation system consists of (1) a *type graph* to define the vocabulary of allowed model elements and their relationships, (2) a set of *constraints* to further restrict the valid models, and (3) a set of *graph transformation rules*. Type graph and constraints can be seen as analogous to the classical, static meta-model.

Thus, a model that conforms to a given (static) meta-model is represented as an *instance graph* of the type graph. One can think of the type graph as a UML class diagram and of the instance graph as a corresponding UML object diagram conforming to the types and constraints of the class diagram.

In the case of dynamic systems evolving at run-time, a single instance graph models the system state at a certain point in time only. For also modeling system evolutions, the dynamic meta-model provides graph transformation rules. These are executable specifications that can be used to define local transformations on graphs. Since graphs represent system states, the transformation rules specify, e.g., possible computation, communication, or reconfiguration operations which can be applied to individual states yielding transitions to new states. Based on individual transformation steps, we can explain, simulate, and analyze the behavioral semantics of dynamic models. In particular, we can generate a state transition system that reflects all reachable states of the system with transitions defined by possible transformation steps.

We provide different meta-models for different levels of abstraction. Thus, for refining an abstract model into a more concrete one, we build on a refinement relationship between the meta-models involved. Formally, this relationship is defined by means of an *abstraction function*, as explained in Section 2. Abstraction is a mapping associating with each concrete model a corresponding abstract model, usually by some kind of projection. Based on this, we can check if a concrete model preserves the *structure* of an abstract model.

In Section 3, we provide conditions for what it means to also preserve the *behavior* of an abstract model. We require that the behavior of the abstract model can be simulated at the concrete level, and we discuss how this property can be checked by model checking at the concrete level. For this purpose, we introduce a translation function, contravariant to abstraction, which maps abstract model properties to the concrete level.

2 Structural refinement

A dynamic meta-model is represented as a typed graph transformation system $\mathcal{G} = \langle TG, C, R \rangle$ consisting of a type graph TG, a set of structural constraints C over TG, and a set R of graph transformation rules $r : L \Rightarrow R$ over TG. The set of valid instance graphs typed over TG is called $\mathbf{Graph}_{TG}$.

Like in a previous paper [1], we exemplify this technique by defining architectural styles as meta-models for software architectures: Graph-based models of a software architecture have to conform to a meta-model representing the underlying architectural style. We consider architectural styles as conceptual models of platforms that systems are implemented on. Graph transformation rules specifying the dynamics of a style capture the reconfiguration and communication mechanisms which allow an architecture to evolve at run-time, supported by the respective platform. We will come back to the dynamic aspect in Section 3.

For now, consider a model-driven development process starting with an abstract, business requirements-driven architecture model of a software system which is shall be refined into a concrete, platform-specific one. In our simplified example, we assume a component-based architectural style for the platform-independent level where components interact through ports that can only be connected if the provided and required interfaces match.

For the platform-specific level, we assume a style that represents *service-oriented architectures (SOA)*. In SOA, the functionality of components is published as *services* to *service requesters*. Special third-party components, called *discovery agencies*, realize service discovery at run-time, i.e., service provider and requester do not need to know each other in advance. For this purpose, the service-providing component has to publish a description of the provided interface to the discovery agency. A service-requesting component can then use the lookup mechanisms of the discovery agency to find suitable service descriptions for its own requirements.

We do not present the type graphs of these two architectural styles; they can be found in [2]. Basically, they define node and edge types for the architectural concepts summarized above. Instance graphs of these type graphs are used to represent platform-independent or service-oriented architectures respectively. For the sake of readability, we use a UML 2.0-like concrete syntax as shown in Fig. 1. The example describes the architecture of an electronic travel agency application. It requests airline systems to book flights for journeys its clients want to purchase.

Given an abstract transformation system $\mathcal{G} = \langle TG, C, R \rangle$ like the platform-independent architectural style and a concrete transformation system $\mathcal{G}' = \langle TG', C', R' \rangle$ like the service-oriented architectural style, structural refinement establishes a relation between abstract instance graphs $G \in \mathbf{Graph}_{TG}$ and concrete instance graphs $G' \in \mathbf{Graph}_{TG'}$: We require that, in order to be a valid refinement of abstract G, concrete G' has to preserve the structure of the abstract graph.

Since the two instance graphs which shall be compared are expressed over different type graphs, this condition is expressed modulo an abstraction function $abs : \mathbf{Graph}_{TG'} \to \mathbf{Graph}_{TG}$ that is assumed to be given together with the type graphs, formally: G' is a structural refinement of G if $G \subseteq abs(G')$.

Figure 1 exemplifies the abstraction function applied to the concrete,

SOA-specific model of the travel agency system (bottom) yielding the abstract, platform-independent model (top). The abstraction removes all platform-specific elements like the discovery component and the service description and requirements documents. Moreover, the platform-specific stereotype ≪service≫ is adapted to the platform-independent vocabulary ≪component≫.

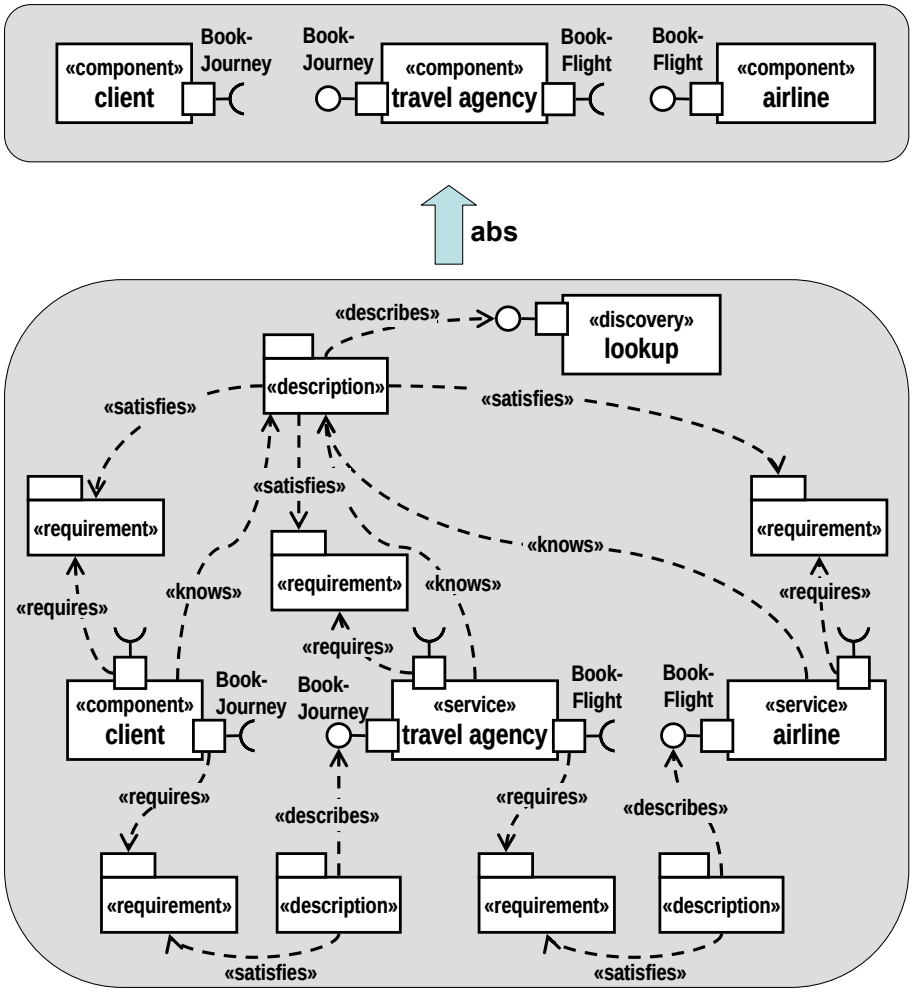


Fig. 1. Abstraction from service-oriented to platform-independent style

There is a range of possibilities for the definition of abstraction functions, from a simple mapping between the two type graphs which can be lifted to instance graphs by renaming the types of the graph elements (cf. [2]) to complex mappings defined by transformation rules, e.g., in order to detect design patterns in reverse engineering. Rather than fixing one concrete way of definition, in this paper we will axiomatize the relevant properties of such mappings.

3 Behavioral refinement

The behavioral part of a dynamic model is defined by the graph transformation rules of its meta-model. For instance, for the abstract, component-based architectural style, we assume that components can dynamically bind to provided interfaces at run-time. This can be realized by appropriate reconfiguration operations for interface binding and unbinding as shown in Fig. 2. In this case, the two transformation rules that define the desired bind and unbind operations are symmetric.

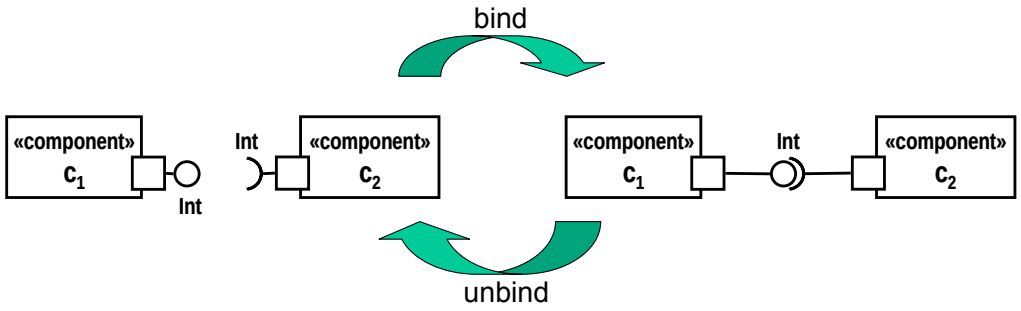


Fig. 2. Reconfiguration rules of abstract architectural style

Formally, the transformation rules are expressed by pairs of instance graphs over the underlying type graph. However, for space reasons and for the sake of better readability, we present them in a UML-like syntax, similar to the instance graphs in the previous section.

Behavior is represented by transitions between instance graphs. The space of possible behaviors is thus given by a *transition system* whose states are the reachable graphs and whose transitions are generated by rule applications. Given the initial state of the model as a start graph, one can generate and explore the transition system by continuously applying transformation rules to previously generated states. To give an example, Figure 3 shows the transition system for the travel agency system in the abstract architectural style. The transitions are labeled by the names of the applied transformation rules. Recently, the automated generation of transition systems from graph transformation system is supported by tools like GROOVE [7] or CheckVML [8].

Similar to the platform-independent style, there are also graph transformation rules in the service-oriented architectural style. However, they have to account for platform-specific restrictions. In the SOA case, for instance, it is required to know the description of a service before it is possible to access it. Therefore, the corresponding reconfiguration rule bind, shown in Fig. 4, includes this additional precondition on its left-hand side. Thus, the bind-operation can only be applied if the service description is known to the component playing the role of the service requester. This is represented by the UML dependency with sterotype ≪knows≫.

The service-oriented style contains further platform-specific transformation rules publish and find, which enable dynamic service discovery by publishing

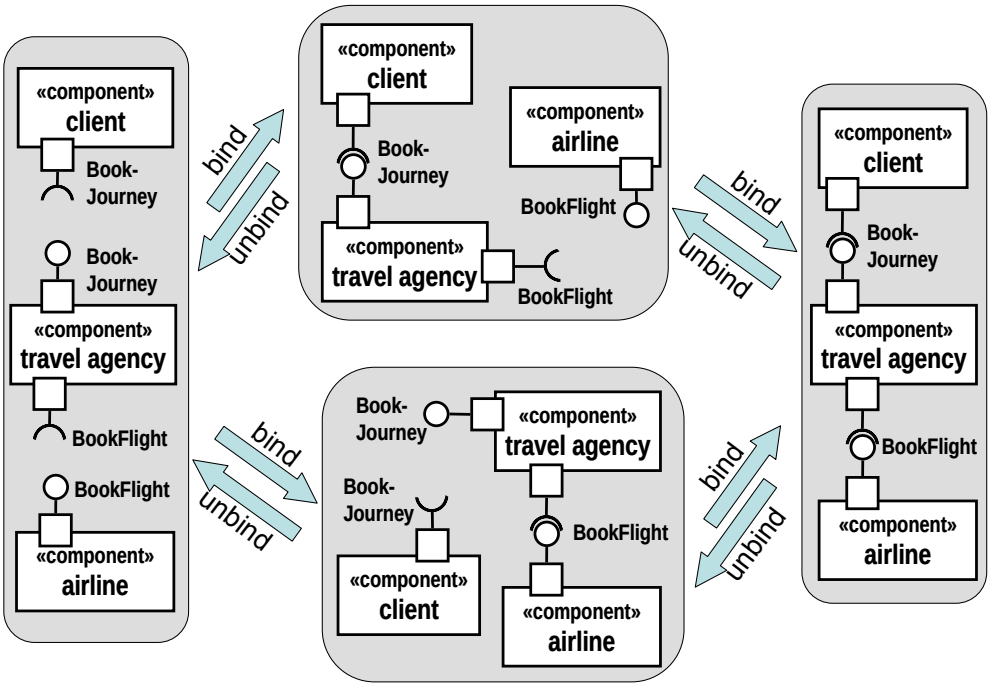


Fig. 3. Transition system for the abstract travel agency architecture

service descriptions to discovery agencies and by querying suitable descriptions that satisfy certain requirements. These operations might be required before a bind-operation can be performed. Due to space limitations, the rules presented in Fig. 4 form a simplified version of the SOA style presented in [1,2].

Like at the platform-independent level, we can now apply the SOA rules to the SOA-specific variant of the travel agency architecture (see Fig. 1), yielding another transition system which represents the platform-specific behavior.

For checking the behavioral refinement of two models (architectures), we now have to take into account the transition systems that can be generated within the underlying dynamic meta-model (architectural style). Formally, we consider again an instance graph G of an abstract system $\mathcal{G} = \langle TG, C, R \rangle$ and an instance graph G' of a concrete system $\mathcal{G}' = \langle TG', C', R' \rangle$. We assume that G' represents a *structural* refinement of G. In order to be a *behavioral* refinement, the behavior of G' must refine the behavior of G. This is the case if every path $G \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n$ in the abstract transition system has a correspondent path $G' \stackrel{*}{\Longrightarrow} G'_1 \stackrel{*}{\Longrightarrow} \ldots \stackrel{*}{\Longrightarrow} G'_n$ in the concrete transition system with G'_i refining G_i (that is, $G_i \subseteq abs(G'_i)$) for all $i = 1 \ldots n$.

Each step in the abstract system can be matched by a sequence of steps in the concrete system. A single transformation *step* $G_i \Rightarrow G_{i+1}$ of the abstract path is refined by a transformation *sequence* $G'_i \stackrel{*}{\Longrightarrow} G'_{i+1}$ at the concrete level because it might be necessary to perform a set of consecutive concrete steps in order to realize the abstract one (for example, additional publish and find operations in the SOA case).

Building on the model checking approaches for graph transformation sys-

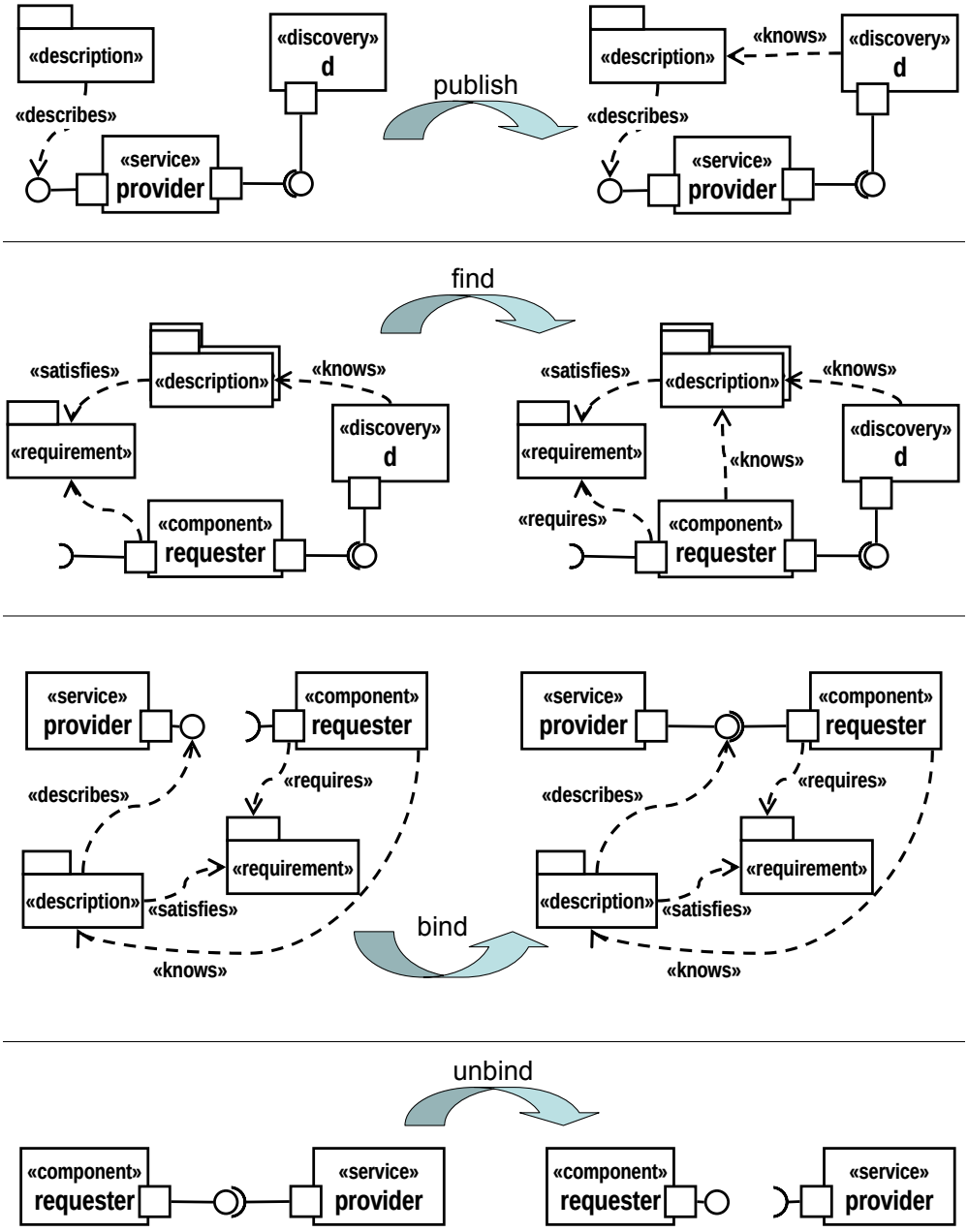


Fig. 4. Reconfiguration rules of the service-oriented architectural style

tems mentioned above, we would like to formulate the refinement of an abstract path as a reachability problem in the concrete transition system. However, the condition for behavior refinement includes the structural refinement of G_i by G'_i which, in general, requires to project the concrete graph to the abstract level in order to verify the desired inclusion.

In order to express the same property solely at the level of the concrete system, we must assume a second mapping $trans : \mathbf{Graph}_{TG} \rightarrow \mathbf{Graph}_{TG'}$, contravariant to abstraction. It translates an abstract instance graph into a

concrete one representing the reformulation of the abstract state over the concrete type system. Note that the concrete graph does not necessarily represent a complete state of the concrete model, but rather a minimal *pattern* which has to be present in order for the requirements of the abstract graph to be fulfilled. Thus, we consider a concrete instance graph as a valid refinement of an abstract one if it contains this pattern as a subgraph, formally $trans(G) \subseteq G'$.

For example, Fig. 5 shows how the platform-independent model of the travel system is translated into a pattern for the service-oriented style with services instead of components where desired. According to the definition above, a valid service-oriented architecture containing this pattern, e.g., the SOA model at the bottom of Fig. 1, is a refinement of the abstract model.

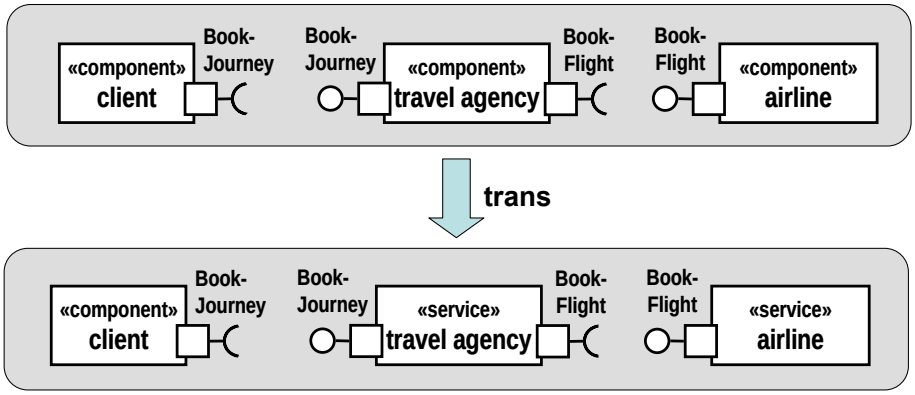


Fig. 5. Translation from platform-independent to service-oriented style

To make sure that the above condition is equivalent to the original one for structural refinement, we have to ensure the mutual consistency of the two contravariant mappings. This is formally expressed as a *satisfaction condition*, reminiscent of similar conditions in algebraic specification or logics, i.e.,

$$trans(G) \subseteq G' \text{ iff } G \subseteq abs(G').$$

In this case, we say that the two mappings are compatible.

Under this assumption, refinement can be formulated as follows. Concrete graph G' *refines* abstract graph G if

- $trans(G) \subseteq G'$
- for every transformation step $G \Rightarrow H$ in the abstract system there exists a transformation *sequence* $G' \stackrel{*}{\Longrightarrow} H'$ such that H' refines H.

It follows from the satisfaction condition that the first clause above is equivalent to the original condition $G \subseteq abs(G')$, expressed in terms of abstraction. However, the new condition can be verified solely at the concrete level.

The second clause is effectively a co-inductive definition of a simulation relation. Spelled out in terms of sequences, it says that for every (possibly infinite) path $G \Rightarrow G_1 \Rightarrow G_2 \Rightarrow \ldots$ in the abstract system there exists a path $G' \stackrel{*}{\Longrightarrow} G'_1 \stackrel{*}{\Longrightarrow} G'_2 \stackrel{*}{\Longrightarrow} \ldots$ in the concrete system with $trans(G_i) \subseteq G'_i$.

4 Related work

The use of meta-models for defining graphical languages has become very popular in the context of the *Meta-Object-Facility (MOF)* authored by the OMG. They also define meta-models as type graphs with additional constraints like, e.g., cardinalities. A model is an instance of the meta-model, if it conforms to the type graph.

In our work, we extend the static declaration of the meta-model by graph transformation rules which allow the definition of dynamic model evolutions as a simulation of system evolution. The use of graph transformation techniques to capture dynamic semantics of models has been inspired by Engels et al. in [4]. That approach extends meta-models defining the abstract syntax of a modeling language like UML by graph transformation rules for describing changes to object graphs representing the states of a model.

In [2], we have already considered several levels of platform abstraction that allow an MDA-like refinement from platform-independent architectures to more platform-specific ones. This has brought us to the question of suitable notions for refining graphs and graph transformation behavior: While structural refinement implies a relation between the involved type graphs, the idea for behavioral refinement is to relate the transformation rules of the involved graph transformation systems. In general, one can place these refinement relationships in a continuum from syntactic to semantically defined relations.

Große-Rhode et. al. [5], for instance, propose a refinement relationship between abstract and concrete rules that can be checked syntactically. One of the conditions requires that, e.g., the abstract rule and its refinement must have the same pre- and post-conditions. Based on this restrictive definition they can prove that the application of a concrete rule expression yields the same behavior as the corresponding abstract rule. The draw-back of this approach is that it cannot handle those cases where the refining rule expression should have additional effects on elements of the concrete level that do not occur in the abstract rule. And, the approach does not allow for alternative refinements of the same abstract rule depending on the context of its application.

Similarly, the work by Heckel et. al. [6] is based on a syntactical relationship between two graph transformation systems. Although this approach is less restrictive as it allows additional elements at the concrete level, it is still difficult to apply if there are no direct correspondences between abstract and concrete rules. Moreover, their objective is to project any given concrete transformation behavior to the abstract level, and not vice versa as in our case. Thus, refinement means a restriction of behavior rather than its extension.

In our work, we propose a more flexible, semantically defined notion of refinement. We do not require a fixed relation between transformation rules but only between the structural parts of the graph transformation system. Then, we check whether selected system states in the abstract system are also reachable at the concrete level, no matter by which sequence of transforma-

tions. By avoiding the functional mapping between rules, we can also relate transformation systems with completely different behavior, and we are flexible enough to cope with alternative refinements.

5 Conclusion

We have discussed semantic conditions for the refinement of dynamic models expressed as instances of graph transformation systems. Applications of this technique include so far the refinement of architectural models based on corresponding relations between architectural styles.

We are planning to support the approach by a coupling of CASE tools with editors and analysis for graph transformation systems, presently conducting experiments with existing model checkers.

References

[1] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modeling and validation of service-oriented architectures: Application vs. style. In *Proc. European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 03*, pages 68–77. ACM Press, 2003.

[2] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based refinement of dynamic software architectures. In *Proc.* 4^{th} *Working IEEE/IFIP Conference on Software Architecture, WICSA4*, pages 155–164. IEEE, 2004.

[3] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–265, 1996.

[4] G. Engels, J.H. Hausmann, R. Heckel, and St. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In *Proc. UML 2000 - The Unified Modeling Language*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.

[5] M. Große-Rhode, F. Parisi Presicce, and M. Simeoni. Spatial and temporal refinement of typed graph transformation systems. In *Proc. Math. Foundations of Comp. Science 1998*, volume 1450 of *LNCS*, pages 553–561. Springer, 1998.

[6] R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of typed graph transformation systems. *Math. Struct. in Computer Science*, 6:613–648, 1996.

[7] A. Rensink. The GROOVE simulator: A tool for state space generation. In M. Nagl and J. Pfalz, editors, *Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE '03)*, LNCS. Springer, 2003. To be published.

[8] D. Varró. Towards symbolic analysis of visual modeling languages. In *Proc. GT-VMT 2002 - Int. Workshop on Graph Transformation and Visual Modeling Techniques*, volume 72 of *ENTCS*, pages 57–70. Elsevier, 2002.

Detecting Structural Refactoring Conflicts Using Critical Pair Analysis

Tom Mens [1]

Software Engineering Lab
Université de Mons-Hainaut
B-7000 Mons, Belgium

Gabriele Taentzer and Olga Runge [2]

Technische Universität Berlin
D-10587 Berlin, Germany

Abstract

Refactorings are program transformations that improve the software structure while preserving the external behaviour. In spite of this very useful property, refactorings can still give rise to structural conflicts when parallel evolutions to the same software are made by different developers. This paper explores this problem of structural evolution conflicts in a formal way by using graph transformation and critical pair analysis. Based on experiments carried out in the graph transformation tool AGG, we show how this formalism can be exploited to detect and resolve refactoring conflicts.

Key words: refactoring, restructuring, graph transformation, critical pair analysis, evolution conflicts, parallel changes

1 Introduction

Refactoring is a commonly accepted technique to improve the structure of object-oriented software [2]. Nevertheless, there are still a number of problems if we want to apply this technique in a collaborative setting, where different software developers can make changes to the software in parallel.

To illustrate these problems, consider the scenario of a large software development team, where two developers independently decide to refactor the same software. It is possible that these parallel refactorings are incompatible,

[1] Email:tom.mens@umh.ac.be
[2] Email:gabi@cs.tu-berlin.de

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

in the sense that they cannot be combined together. As an example, assume that a *Move Variable* refactoring and an *Encapsulate Variable* refactoring are applied in parallel to the same variable in the same class. Both refactorings are clearly in conflict since they cannot be serialised as they both affect the same variable in different incompatible ways.

It is also possible that two parallel refactorings can only be combined in a particular order. As an example, assume that a *Rename Variable* refactoring and an *Encapsulate Variable* refactoring are applied in parallel to the same variable in the same class. One can decide to rename the variable first, and then encapsulate it, but not the other way round. The reason is that the encapsulation introduces an auxiliary setter and getter method whose names rely on the variable name.

To address the problems illustrated above, we propose to take a formal approach based on *graph transformation* and *critical pair analysis* [1,4,5]. We will perform a feasibility study using the AGG tool. As such, the contribution of our paper will be twofold:

- to show the feasibility of the technique of critical pair analysis for a new practical application;
- to support refactoring tool developers with a formal means to analyse the consistency of refactoring suites, and to allow them to identify unanticipated dependencies between pairs of refactorings.

2 The AGG tool

We decided to use the tool AGG (see `http://tfs.cs.tu-berlin.de/agg`) for our experiments. It is the only graph transformation tool we are aware of that supports critical pair analysis, a crucial ingredient of our approach towards the detection of refactoring conflicts.

2.1 Specifying graph transformations

To reason about object-oriented software evolution, we specify object-oriented programs as graphs, that have to respect the constraints specified by a *type graph*. This type graph acts as an object-oriented metamodel. The metamodel we expressed in AGG is shown in Figure 1. It expresses the basic object-oriented concepts (such as classes, methods and variables), their attributes (such as name and visibility), and their relationships (such as inheritance, containment and typing) with associated multiplicities. We deliberately decided to use this simple metamodel instead of a full-fledged one, because our goal was to perform a feasibility study.

Representative refactorings are expressed as graph transformations using this metamodel. A *graph transformation* $t : G \xrightarrow[p(m)]{} H$ is defined as a pair consisting of a *graph transformation rule* $p : L \to R$ and a *match* $m : L \to G$.

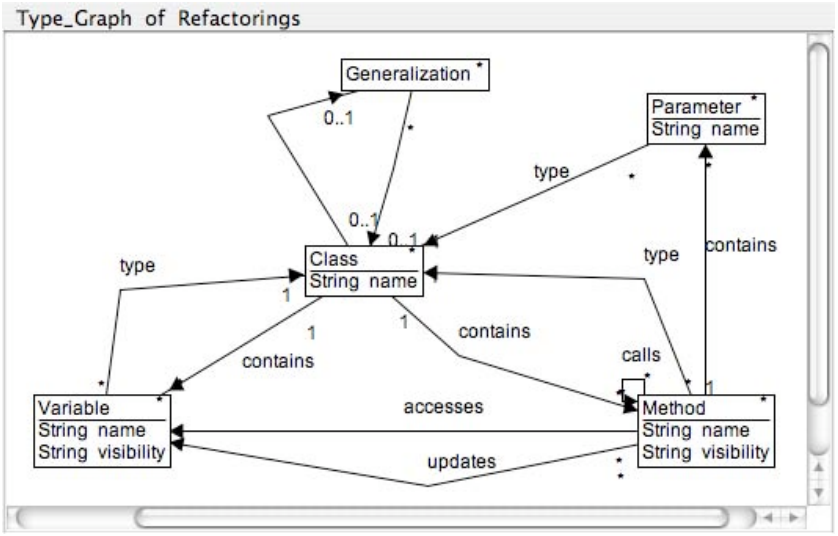


Fig. 1. Type graph representing the object-oriented metamodel.

The rule p specifies how its left-hand side (LHS) L has to be transformed into its right-hand side (RHS) R. The match m specifies an occurrence of this LHS in the graph that needs to be transformed. Note that there may be more than one possible match. As shown in [5], one can easily extend this definition to come to a notion of *typed graph transformations* that respect the type constraints imposed by the type graph.

As a concrete example, the transformation *Encapsulate Variable* in Figure 2 can be applied to a class containing a variable of a particular type. After the transformation, a setter method and getter method are added to the class, but the rest of the structure is preserved. This is visualised by assigning numbers 1 to 5 to nodes and edges in the LHS and RHS. Nodes and edges that have the same number in the LHS and RHS are preserved by the transformation. All nodes and edges in the RHS that do not have a number assigned (such as the setter and getter method) are newly introduced.

Note that the graphs we use are attributed, i.e., the nodes in the graph may contain attributes whose values may be modified by the transformation. This is for example the case in Figure 2 with the attribute `visibility` of variable node `1`, whose value is modified from `public` to `private`.

Another useful feature of AGG is the possibility to specify *negative application conditions* (NACs) [3] that capture the preconditions of a transformation. These NACs can be considered as a kind of forbidden subgraphs. For example, the NAC *No Setter* for transformation rule *Encapsulate Variable* in Figure 2 expresses that the class containing the variable to be refactored must not contain a setter method for this variable, since this method will be added by the transformation. To express this, we need to specify an attribute condition which relates the name of the method in the NAC to the corresponding one in the RHS. The rule *Encapsulate Variable* contains a second NAC that forbids the existence of a getter method for the variable to be encapsulated.

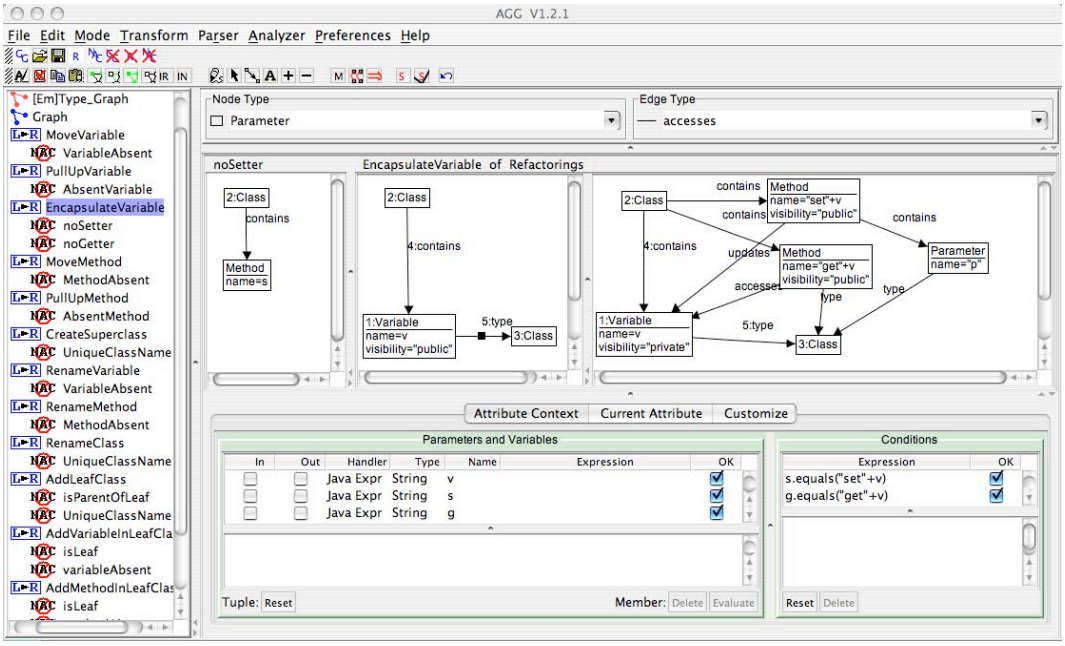


Fig. 2. The AGG tool in action. In the left pane, all refactorings specified as graph transformations are listed, together with their NACs. On the right of it, the specification of the *Encapsulate Variable* refactoring is given as a graph transformation rule with a NAC *No Setter*, a left-hand side, and a right-hand side. The Attribute Context for the Method attribute `name` in the bottom panes specifies the additional relation that its value `s` in the NAC must be equal to `"set"+v` in the RHS.

2.2 Critical pair analysis

Critical pair analysis was first introduced in term rewriting, and has been generalized to graph rewriting later. A critical pair formalises the idea of a minimal example of a potentially conflicting situation. Given two transformations $t_1 : G \xrightarrow[p_1(m_1)]{} H_1$ and $t_2 : G \xrightarrow[p_2(m_2)]{} H_2$, t_1 has an *asymmetric conflict* with t_2 if it can be performed before, but not after t_2. If the two transformations disable each other in any order, they have a *symmetric conflict*.

The reasons why rule applications can be conflicting are threefold:

(i) One rule application deletes a graph object which is in the match of another rule application.

(ii) One rule application generates graph objects that give rise to a graph structure that is prohibited by a NAC of another rule application.

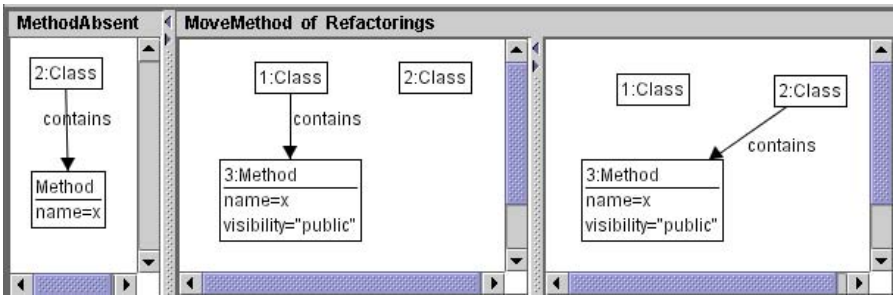
(iii) One rule application changes attributes being in the match of another rule application.

To find all conflicting rule applications, minimal critical graphs are computed to which rules can be applied in a conflicting way. Basically we consider all overlapping graphs of the left-hand sides of two rules with the obvious matches and analyze these rule applications. All conflicting rule applications

thus found are called *critical pairs*. If one of the rules contains NACs, the overlapping graphs of one LHS with a part of the NAC have to be considered in addition.

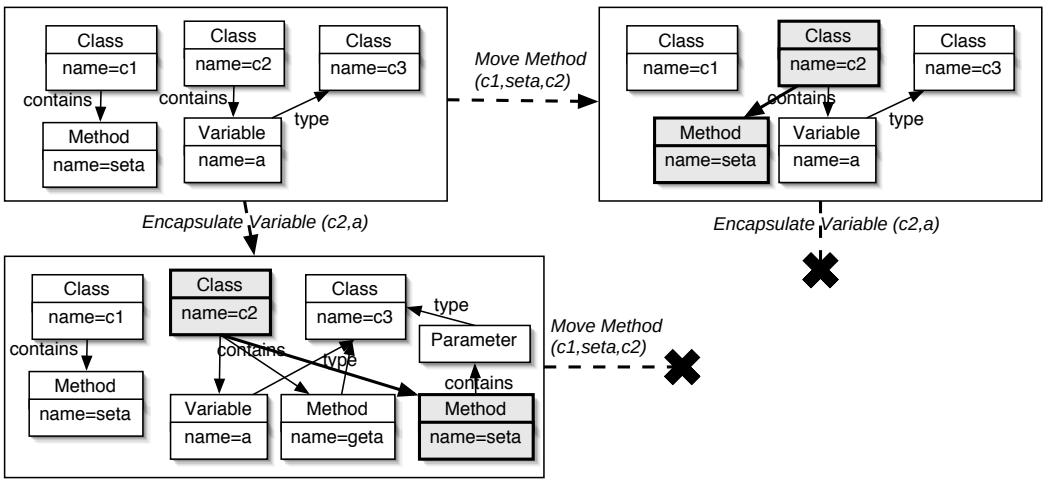
AGG supports the critical pair analysis for typed attributed graph transformations. Given a set of graph transformation rules, it computes a table which shows the number of critical pairs for each pair of rules. The number of detected critical pairs for transformation rules can be reduced drastically if there is a type graph with multiplicity constraints (as in Figure 1). Upper bounds of the multiplicity constraints are then used to reduce the set of critical pairs by throwing out the meaningless ones.

Fig. 3. Graph transformation rule for *Move Method*.



As a concrete example, let us compute the critical pairs between the graph transformation rules *Encapsulate Variable* (of Figure 2) and *Move Method* (shown in Figure 3). There is a symmetric conflict between both rules, and the number of computed critical pairs in both cases is 2. Figure 4 illustrates this graphically.

Fig. 4. Example of a symmetric conflict between graph transformations *Move Method* and *Encapsulate Variable*



If we move a method to a class in which we want to encapsulate a variable afterwards, there is a first critical pair that represents the conflict that the name of the method that is moved coincides with the name of the setter method that needs to be introduced by *Encapsulate Variable*. The second critical pair, which is very similar, represents a name conflict with the getter method.

The other way around, if we first apply the *Encapsulate Variable* transformation, we get a similar situation. *Move Method* cannot be applied when the method needs to be moved to the class of the encapsulated variable, and the method name coincides with either the name of the setter method or the name of the getter method.

3 Specification of refactorings

To be able to detect conflicts between refactorings applied in parallel by different software developers, we specified some representative refactorings identified by Fowler [2] as typed attributed graph transformations. The preconditions of the refactorings were directly expressed as negative application conditions on the graph transformations.

- *Encapsulate Variable* takes a public variable in a class and replaces it by a private variable with two public accessor methods. One for getting the value of the variable, and one for setting its value. The graph transformation rule for this particular refactoring is shown in Figure 2;
- *Move Method* moves a public method from a class to another class, not necessarily belonging to the same inheritance hierarchy. The graph transformation rule is shown in Figure 3.
- *Move Variable* moves a public variable from a class to another class, not necessarily belonging to the same inheritance hierarchy. The graph transformation rule is very similar to the one for *Move Method*.
- *Pull Up Variable* moves a public or protected variable from a class to a superclass that resides one level up the inheritance hierarchy. The graph transformation rule is shown in Figure 5.
- *Pull Up Method* moves a public or protected method from a class to a superclass that resides one level up the inheritance hierarchy. The graph transformation rule is similar to the one for *Pull Up Variable*.
- *Create Superclass* creates an intermediate abstract superclass for a given class. The graph transformation rule is shown in Figure 6.
- *Rename Method* changes the name of a method in a class to a new one which is unique within this class. The graph transformation rule is shown in Figure 7.
- *Rename Variable* changes the name of a variable in a class to a new one which is unique within this class. The graph transformation rule is similar

to the one for *Rename Method*.

- *Rename Class* changes the name of a class to a new unique name. The graph transformation rule is similar to the one for *Rename Method*.

Fig. 5. Graph transformation rule for *Pull Up Variable*.

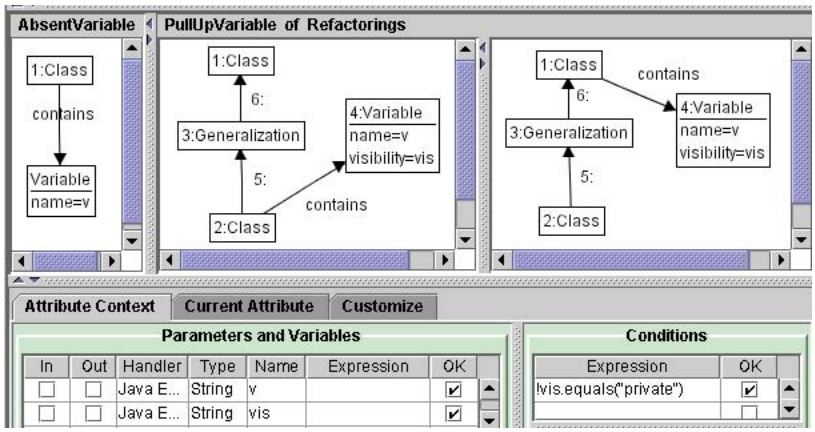
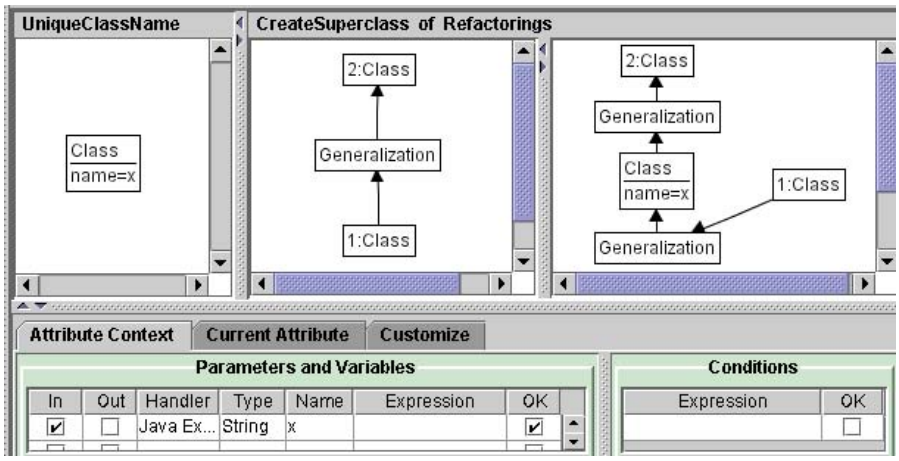
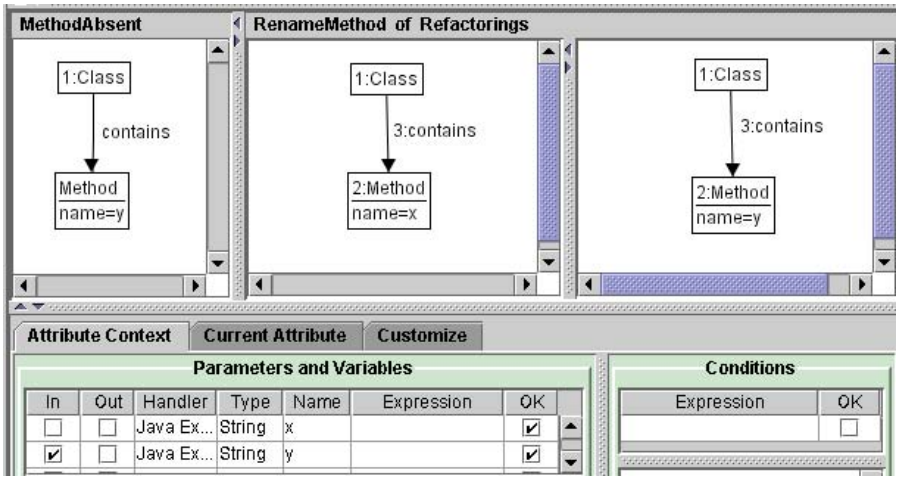


Fig. 6. Graph transformation rule for *Create Superclass*.



One should note that we deliberately did not implement all details of each refactoring in our graph transformations, since it was not our intent to build a full-fledged refactoring tool, but rather to perform a feasibility study that would show that the most important conflicts between parallel refactorings can be detected by critical pair analysis. For example, we decided to restrict *Create Superclass*, *Pull Up Variable* and *Pull Up Method* to a single subclass rather than a set of subclasses. We also did not express all necessary preconditions for each refactoring, as this would only make the analysis more difficult and computation intensive. Although, in theory, these simplifications may lead to false negatives during conflict detection, in practice, it turned out that all of

Fig. 7. Graph transformation rule for *Rename Method*.



the conflicts we expected to occur were actually detected, as we will show in the next section.

4 Analysis of refactoring conflicts

We applied the critical pair analysis algorithm of AGG to our selection of 9 representative refactorings. We observed that, for many pairs of refactorings, duplicate critical pairs were reported for the same conflict. We even found some bugs in the initial critical pair analysis algorithm. Therefore, we improved the algorithm so that it reports only those critical pairs that actually correspond to distinct conflicts. The results of this improved algorithm are shown in Figure 8. All critical pairs can be considered in detail on the AGG Web page.

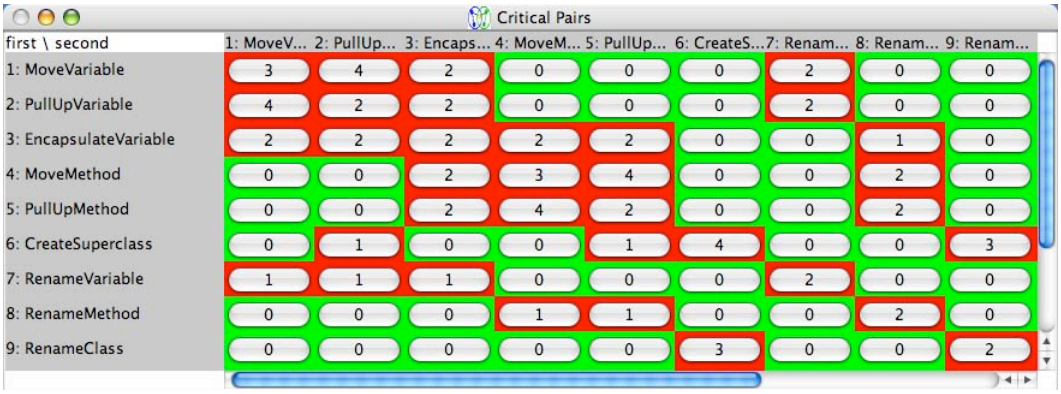


Fig. 8. Critical pair analysis of the refactoring transformations.

The obtained results correspond to what we expected. For example, we

expected a certain similarity between the conflicts generated by *Move Method* and *Pull Up Method* (resp. *Move Variable* and *Pull Up Variable*) since they both move a method (resp. variable) to another location. We also expected similar conflicts for *Move Variable* and *Move Method*, as well as for *Pull Up Variable* and *Pull Up Method*. Finally, we expected many similarities between *Rename Class*, *Rename Variable* and *Rename Method*.

What follows is a detailed discussion of the analysis we performed on the computed critical pairs. A first observation is that parallel applications of the same rule are always in potential conflict. In other words, the diagonal of the critical pair table always contains critical pairs. The reason for this is given below:

(i) Applying *Move Variable* twice to the same variable means that it should be moved to two different classes which is obviously a conflict. Also, two different variables with the same name cannot be moved to the same class due to the negative application condition. Applying *Move Method* twice generates similar conflicts as applying *Move Variable* twice.

(ii) *Pull Up Variable* is in conflict with itself because it cannot be used to pull up two different variables with the same name to the same class due to the negative application condition. Applying *Pull Up Method* twice generates similar conflicts as applying *Pull Up Variable* twice.

(iii) Applying *Encapsulate Variable* twice generates a conflict because one cannot introduce the same accessor methods twice.

(iv) *Create Superclass* is in conflict with itself, since the generalization relation between the class for which a new superclass must be created and its current superclass is deleted. Stated differently, the introduction of two new superclasses would give rise to a multiple inheritance hierarchy, which is prohibited by the multiplicities imposed in the type graph of Figure 1. Another conflict arises if two superclasses with the same name are inserted.

(v) Applying *Rename Class* twice generates a conflict, if the name of one and the same class is changed twice in a different way. Another conflict occurs, if two different classes are renamed with the same name. Applying *Rename Variable* or *Rename Method* twice generates similar conflicts as applying *Rename Class* twice.

A *symmetric conflict* arises in the following situations:

(i) *Move Variable* and *Pull Up Variable* are in conflict if the same variable is pulled up and moved. Furthermore, pulling up one variable and moving another with the same name into the same class causes a conflict due to the negative application conditions of both rules. *Move Method* versus *Pull Up Method* gives rise to a similar symmetric conflict.

(ii) *Move Variable* versus *Encapsulate Variable* causes a symmetric conflict. After moving a variable, it cannot be encapsulated (within the original

class) anymore. Conversely, encapsulating a variable it is no longer public and cannot moved anymore. *Pull Up Variable* versus *Encapsulate Variable* gives rise to a similar symmetric conflict.

(iii) *Move Method* versus *Encapsulate Variable* generates a symmetric conflict as explained in section 2.2. *Pull Up Method* versus *Encapsulate Variable* gives rise to a similar symmetric conflict.

(iv) *Create Superclass* is in conflict with *Rename Class*, if both rules create a new class with the same name.

(v) *Rename Variable* and *Move Variable* resp. *Pull Up Variable* are in symmetric conflict, since the variable to be moved or pulled up is renamed. Otherwise, the variable to be renamed is moved (pulled up) to another class. The symmetric conflicts between *Rename Method* and *Move Method* resp. *Pull Up Method* are similar.

We encountered *asymmetric conflicts* in the following situations:

(i) *Create Superclass* causes an asymmetric conflict on *Pull Up Variable*, since it modifies the generalization relation needed for pulling up the variable. It causes a similar asymmetric conflict on *Pull Up Method*.

(ii) *Rename Variable* causes an asymmetric conflict on *Encapsulate Variable*, since it renames the variable to be encapsulated.

(iii) *Encapsulate Variable* causes an asymmetric conflict on *Rename Method*, since it creates new methods with names which might be used for renaming.

It is important to stress here that the number of conflicts that are detected by the algorithm relies on the chosen metamodel as well as on the specification of the refactorings. Since we made some simplifications to both in our feasibility study, the number of detected critical pairs is likely to increase if we would apply it to a more realistic refactoring suite.

5 Conflict resolution

Critical pairs describe potential conflicts between different rule applications. Often it is possible to show that this critical situation is confluent. Intuitively, this means that the application of one conflicting rule may prohibit the application of the other one, but further transformations may be applied to resolve the conflicting situation. Formally, a critical pair $(G \to H_1, G \to H_2)$ is confluent if there are transformations $(H_1 \to X, H_2 \to X)$ that lead to the same result graph X.

In the following, we discuss to which extent the potential conflicts found by critical pair analysis are confluent and can thus be resolved. We performed the conflict resolution analysis manually. It is left to future work to automate this analysis in AGG.

We start with explaining all conflicts due to parallel applications of the

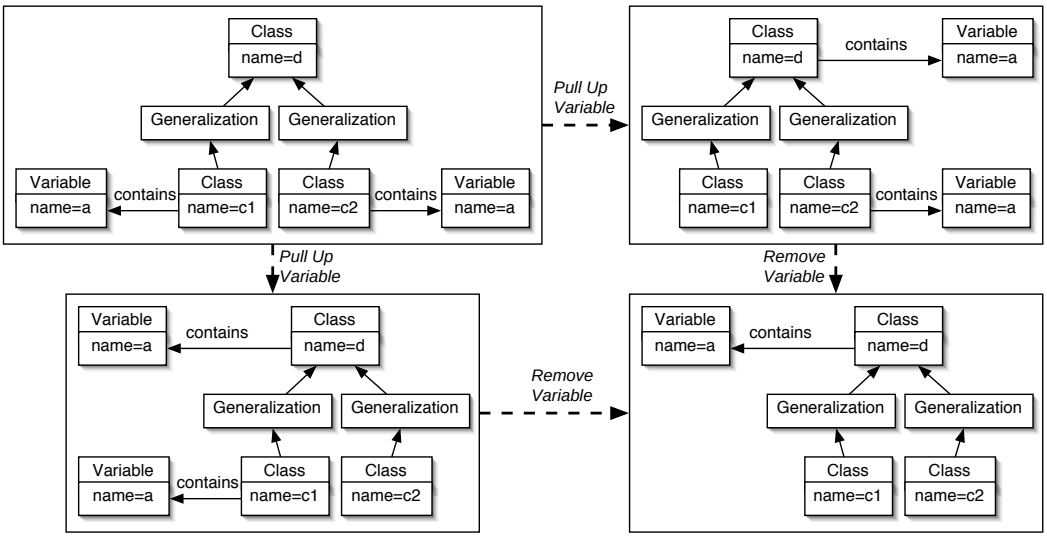


Fig. 9. Resolving parallel evolution conflicts by analysing confluence of critical pairs.

same rule:

(i) Moving a variable first to some class and then to another class leads to a conflict that cannot be solved automatically. One of these moves has to be given the priority by the developer. Trying to move different variables with the same name to the same class also results in a critical pair. It can be solved by renaming one of the variables, i.e., applying rule *Rename Variable* to it, and moving the other variable afterwards. Applying *Move Method* twice generates similar conflicts as applying *Move Variable* twice. Thus, conflict solving is similar.

(ii) If two different variables with the same name (but residing in different subclasses) need to be pulled up into the same class, this conflict can be solved by deleting one of the two variables and pulling the other one up. This solution is visualised in Figure 9. Applying *Pull Up Method* twice generates similar conflicts as applying *Pull Up Variable* twice. Thus, conflict solving is similar.

(iii) Applying *Encapsulate Variable* twice for the same variable needs to be resolved by ignoring one of both rule applications.

(iv) Applying *Create Superclass* twice leads to conflicts that can be resolved by ignoring one of both rule applications.

(v) Renaming a class twice leads to a conflict that cannot be solved automatically. One of these renamings has to be given the priority by the developer. If two different classes with different names should be renamed using the same name, this also results in a critical pair. It can be solved by manually choosing only one of the two classes to be renamed. Applying *Rename Variable* or *Rename Method* twice generates similar conflicts as applying *Rename Class* twice. Thus, conflict solving is similar.

Now, let us see how the symmetric conflicts can be resolved:

(i) Pulling up and moving the same variable is confluent, if the variable is moved to a class that has a superclass. In this case, the variable can still be pulled up after moving. The other way round, the variable can always be moved after pulling it up.
If the variable is moved to a class without superclass, the critical pair is not confluent, because the pull up refactoring cannot be performed (due to absence of the superclass).
A third situation, pulling up and moving two different variables with the same name into the same class causes a confluent conflict situation. It can be solved by renaming first one of the variables and performing the refactoring afterwards.
Move Method versus *Pull Up Method* generates similar conflicts as *Move Variable* and *Pull Up Variable*. Thus, conflict solving is similar.

(ii) *Move Variable* versus *Encapsulate Variable*: Moving first a variable, it can be encapsulated within its new class, thus this situation is confluent. Encapsulating the variable first we reach the same state of changes if afterwards not only the variable is moved, but also the newly created getter and setter methods. These refactorings are only possible, if such accessor methods do not already exist in the new class. Otherwise, additional renamings have to be performed to make the situation confluent.

(iii) *Pull Up Variable* versus *Encapsulate Variable*: If we pull up the variable first, it can be encapsulated within the superclass. If we encapsulate it first, not only the variable but also its accessor methods have to be pulled up (using *Pull Up Method*). Again, as in the previous case, additional renamings may have to be performed to make the situation confluent.

(iv) *Move Method* versus *Encapsulate Variable*: If encapsulating a variable results in the creation of a method with the same name as the method to be moved to the same class, this conflict can be solved by first renaming the method to be moved and then moving it and encapsulating the variable.
Pull Up Method versus *Encapsulate Variable* generate a similar conflict as *Move Method* versus *Encapsulate Variable*. Thus, conflict solving is similar.

(v) Applying *Create Superclass* and *Rename Class* leads to conflicts that cannot be solved automatically. One of these refactorings has to be given the priority.

(vi) *Rename Variable* versus *Move Variable*: Moving a variable first, it has to be renamed within its new class. Renaming it first, the renamed variable is moved.
Pull Up Variable causes a similar conflict on *Rename Variable*. The conflicts between *Rename Method* and *Move Method* resp. *Pull Up Method* are also similar. Thus, conflict solving is similar in all those cases.

Finally, we discuss resolution of the asymmetric conflicts:

(i) Applying *Create Superclass* first *Pull Up Variable* has to be applied twice to get the same effect as pulling first up and then creating a superclass for the subclass. A similar conflict is caused on *Pull Up Method*. Thus, conflict solving is similar.

(ii) *Rename Variable* versus *Encapsulate Variable*: Renaming a variable first, the encapsulation has to be done on the renamed variable. The same effect is obtained by encapsulating first and renaming then not only the variable, but also its accessor methods.

(iii) *Rename Method* versus *Encapsulate Variable*: Encapsulating a variable first a new method is created. If a method is renamed to the name of this new method, this causes a conflict that needs to be resolved by ignoring one of the refactorings, or by performing an additional renaming.

6 Related work

In [5], the formalism of critical pairs was explained and related to the formal property of confluence of typed attributed graph transformations. In [4], critical pair analysis is used to detect conflicting requirements in independently developed use case models. In [1], critical pair analysis has been used to increase the efficiency of parsing visual languages by delaying conflicting rules as far as possible.

The problem that has been addressed in this paper is a well-known problem in the context of version management, and is referred to as software merging [7]. Two other approaches that rely on graph transformation to tackle the problem of software merging were proposed by Westfechtel [13] and Mens [6]. Like our approach, they attempt to detect structural merge conflicts. The novel contribution of the current paper, however, is the use of critical pair analysis to address this problem.

Refactoring is also a very active research domain [9]. Formal approaches have mainly been used to prove that refactorings preserve the behaviour of the program. Graph transformations have also been used to express refactorings [8,12]. To our knowledge, no formal attempt has been made to detect conflicts between refactorings applied in parallel.

7 Discussion

In this paper, we explored the problem of detecting and resolving structural conflicts that arise due to parallel evolution. We expressed refactorings as typed attributed graph transformations with negative application conditions, we used critical pair analysis to detect evolution conflicts, and confluence analysis to resolve the conflicts. From a practical point of view, the feasibility study we performed already provided very useful results. It allowed us to gain

insight in the similarities of, and interactions between, different refactorings.

We believe that our approach has a lot of potential, and requires further exploration. For example, our approach may be very beneficial for refactoring tool developers. [11,10] proposed to combine the detection of "code smells" with a refactoring engine that resolves these smells. For each detected smell, there are typically many different refactorings that can be applied to resolve them [2], and some of these refactorings may be in conflict. Hence, a critical pair analysis of the possible choices may help the programmer to decide which refactoring to apply.

Another interesting application would be to incorporate conflict resolution strategies (based on confluence analysis) into refactoring tools. Suppose that a user wants to apply two refactorings sequentially, but the second one is not applicable due to a critical pair conflict. Rather than simply refusing to apply the second refactoring, the tool could suggest to perform an automatic resolution of the conflict that enables to apply the second refactoring.

During our experiments with AGG we encountered a number of limitations, which required us to improve the critical pair analysis algorithm. In the new version of AGG that we developed, the preparation of the critical pairs is already quite user-friendly, but there is still a potential for improvement to better understand the critical situations.

Another problem we have to deal with is the presence of *false positives* and *false negatives*. In order to reduce the possibility of *false negatives*, one needs to provide a more complex metamodel and more realistic refactorings. *False positives* arose because our transformation rules did not take the transitive closure of the specialization hierarchy into account. A straightforward solution would be to add specific transformation rules that compute the transitive closure before actually applying the refactoring rules. An alternative solution would be to use path expressions, but this would be very difficult to implement in AGG due to inherent limitations in the underlying formal approach.

References

[1] Bottoni, P., G. Taentzer and A. Schürr, *Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation*, in: *Proc. IEEE Symp. Visual Languages*, 2000.

[2] Fowler, M., "Refactoring: Improving the Design of Existing Programs," Addison-Wesley, 1999.

[3] Habel, A., R. Heckel and G. Taentzer, *Graph Grammars with Negative Application Conditions*, Special issue of Fundamenta Informaticae **26** (1996).

[4] Hausmann, J. H., R. Heckel and G. Taentzer, *Detection of conflicting functional requirements in a use case-driven approach*, in: *Proc. Int'l Conf. Software Engineering* (2002).

[5] Heckel, R., J. M. Küster and G. Taentzer, *Confluence of typed attributed graph transformation systems*, 2002.

[6] Mens, T., *Conditional graph rewriting as a domain-independent formalism for software evolution*, in: *Proc. Int'l Conf. Agtive 1999: Applications of Graph Transformations with Industrial Relevance*, Lecture Notes in Computer Science **1779** (2000), pp. 127–143.

[7] Mens, T., *A state-of-the-art survey on software merging*, Transactions on Software Engineering **28** (2002), pp. 449–462.

[8] Mens, T., S. Demeyer and D. Janssens, *Formalising behaviour preserving program transformations*, in: *Graph Transformation*, Lecture Notes in Computer Science **2505** (2002), pp. 286–301, proc. 1st Int'l Conf. Graph Transformation 2002, Barcelona, Spain.

[9] Mens, T. and T. Tourwé, *A survey of software refactoring*, Transactions on Software Engineering **30** (2004), pp. 126–139.

[10] Tourwé, T. and T. Mens, *Identifying refactoring opportunities using logic meta programming*, in: *Proc. 7th European Conf. Software Maintenance and Re-engineering (CSMR 2003)* (2003), pp. 91–100.

[11] van Emden, E. and L. Moonen, *Java quality assurance by detecting code smells*, in: *Proc. 9th Working Conf. Reverse Engineering* (2002), pp. 97–107.

[12] Van Gorp, P., H. Stenten, T. Mens and S. Demeyer, *Towards automating source-consistent UML refactorings*, in: P. Stevens, J. Whittle and G. Booch, editors, *UML 2003 - The Unified Modeling Language*, Lecture Notes in Computer Science **2863** (2003), pp. 144–158.

[13] Westfechtel, B., *Structure-oriented merging of revisions of software documents*, in: *Proc. Int'l Workshop on Software Configuration Management* (1991), pp. 68–79.

Predicting incompatibility of transformations in model-driven development

Mehdi Jazayeri [1] Johann Oberleitner [2]

Technische Universität Wien
Institut für Informationssysteme
Argentinierstraße 8/E1841
A-1040 Wien, Austria

Abstract

The grand vision of model-driven development and model-driven architecture (MDA) is to generate automatically an implementation from a high-level model of the application. The primary ingredients of model-driven development are a platform-independent model (PIM) of the application and a platform-specific model (PSM) which is derived from the PIM for a given target platform. The transformation from PIM to PSM could be done automatically if necessary mappings are defined. Even if this grand vision were to be realized sometime in the future, the evolution of applications developed in this way still poses interesting challenges. We point out specific problems that arise when evolving an application to support different platforms and different technologies. We then propose a supporting tool called a "portability checker" that can help the application developer in evolving applications developed using MDA.

Key words: MDA, Model Transformation, Consistency and Co-Evolution

1 Introduction

One goal of model-driven architecture (MDA) is to support the development of long-lasting systems which evolve gracefully over time. The primary ingredients of model-driven development are a platform-independent model (PIM) of the application and a platform-specific model (PSM) which is derived from the PIM for a given platform, e.g. CORBA, EJB, etc. The transformation from PIM to PSM could be done automatically if necessary mappings are defined. An application developed using this methodology can be evolved over

[1] EMail: jazayeri@infosys.tuwien.ac.at
[2] EMail: joe@infosys.tuwien.ac.at

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

time to run on different and newer platforms by defining new PSMs and new transformations. However, this approach assumes that the PIM is abstract enough that all concepts used in the PIM can be transformed to the new concrete PSMs. There are at least two problems with this assumption. First, the design of PIMs is often the result of many tradeoffs, some of which take into account the capabilities of the anticipated platforms and technologies (i.e. are transactions available? Are entity beans sufficient for the task). Not all these analyses may hold over the lifetime of a system when new platforms and technologies are introduced. It is even possible that a new platform may not support all the features assumed by the PIM. Second, new platforms or technologies are introduced and existing platforms are improved with new features. It is not always a good idea for the PIM to continue using old features when newer, better, ones are available. It is important for the transformation from PIM to PSM to be able to take advantage of these improvements. However, the PIM may not be designed to support such enhancements. In this position paper, we propose a "portability" checker that supports developers in checking if a source model conflicts with features supported by target platforms. With this checker it is possible to verify which parts of a model transformation cannot be done completely if target platforms and technologies are changed. The portability checker marks which sets of source or target model elements cannot be transformed in a new technology platform and provides information which model element pairs map to outdated platform features and therefore should be remodeled. In the following, we explain the different uses for such a tool.

It is clear that model-driven development is still in its infancy and many developments are needed before it becomes a practical methodology. One important requirement is the ability to combine models from different domains, for example, one for the user interface, one for transactions, and another for the execution platform. A model compiler [4] can then transform and combine these models similar to aspect-weaving. Also in this case, the portability checker can be a useful help.

2 Background

This section describes concepts that are important to understand the most important concepts of MDA and the remainder of the paper.

The most important improvements in productivity in software development have been achieved by increasing the level of abstraction[2] including the use of assembler instructions instead of machine code, the use of higher-level programming languages instead of assembler language, and the use of device driver interfaces instead of manually writing device access code.

Model Driven Architecture (MDA) is the OMG's solution to increase the abstraction level one step further to the development of distributed systems. MDA makes intense use of OMG's modeling standards such as UML and MOF

to build models that represent Enterprise Computing systems [6]. These models represent the system independently from a particular middleware platform. Hence, these models are called *Platform Independent Model (PIM)*. A PIM is used to build executable code for a particular middleware system. MDA tools generate this code by generators that use a PIM as input. Often, it is not possible to generate complete executable code directly. Instead, or additionally, a *Platform Specific Model (PSM)* is generated. Mappings from PIM to PSM will allow tools to automatically generate PSM and/or the code. However, if the input model is incomplete, the developers have to manually modify the PSM to have a complete implementation of the platform specific code. MDA works very well for parts of an enterprise application where application logic is simple and can be deduced from the models. Recently, a UML action language has been introduced [4] that allow to specify semantics even for procedures. Furthermore, specialized UML profiles, extensions to UML, have been defined to describe a model, for example of a type of application (e.g. distributed computing) or a type of platform (e.g. EJB or CORBA). Models that conform to such profiles can be used as input to the code generators. Our portability checker also makes use of these UML profiles.

It is important to note that no matter how we try to make models independent of implementation platforms, practical considerations often impose some implementation-dependence in the model. An example in MDA, which is familiar to programmers with ordinary programming languages, is when the designer knows that the code generator is not able to generate efficient code for a particularly high-level construct in the independent model. In that case, the designer modifies the model with implementation- (or generator-) dependent changes to ensure an efficient transformation.

3 A portability checker and its applications

A portability (or in general compatibility) checker takes as input an input model and one or more target models and signals any detected incompatibilities. The tool also uses transformation rules that are given for transforming source models (i.e. PIMs) to target models (i.e. PSMs). Additionally, the checker also uses rules for which mappings are possible and considers constraints between source model elements (e.g. transactions) and target model elements (e.g. events). These mappings are used for determining if a transformation leads to complete target models or complete executable target code. These transformation rules and mapping constraints are part of the configuration of the checker and are not input by the user. Due to space restrictions, we rely on the reader's intuitive understanding of these notions and instead concentrate on the uses of the portability checker.

In the remainder of this section, we list some of the uses of a portability checker in model-driven development. The purpose of these examples is to show where the ideal of model-driven architecture runs into practical problems

and how the portability checker can intervene in the development process to keep the process as close to ideal as possible.

3.1 Developing portable software from scratch

The basic use of the portability checker is in developing the same software for different target platforms. MDA allows the design of one platform-independent model (PIM) and postulates the automatic generation of platform-specific implementations. The PIM is supposed to be the basis for generating the code for all target platforms. This generation is based on different UML profiles for different target platforms. The reason that different UML profiles are needed is that not all platforms support all the same features and it is not at all guaranteed that all the features required by the PIM are supported by the UML profiles. As a result, the automatic generation will fail. The designer can use the portability checker at the time that the PIM is being developed to ensure that the PIM features match the support provided by the desired target platforms described by the UML profiles. The incompatibilities detected by the portability checker are brought to the attention of the designer and can be handled in different ways. The designer can redesign the model to support all target platforms or accept the incompatibilities and support different functionality on different platforms or manually implement the problematic code parts. This approach can be used iteratively to deal with all problematic locations. In addition to changing the source model the developer might be able to extend and adapt the code generator to support required features in a target platform.

Although it is unlikely that an entire distributed system is implemented for two target platforms such as CORBA and EJB simultaneously from scratch, it is certainly a common case for designers building components or sub-systems.

3.2 Porting an existing MDA based system

A more likely scenario in model-driven development than starting from scratch is when an existing MDA-based system is to be ported to another platform. This is a typical evolution scenario. In the simple case, we use the portability checker to discover the incompatibilities of the PIM with the new platform. Again we have the same option as before: modify the PIM and adapt the existing implementation also or manually adjust the new platform's implementation. A more difficult case, however, is if the new platform's application has to be developed in parallel with maintaining the existing application. In this case, new functionality has to be introduced in the two platforms simultaneously. The development of the new enhancement must follow the process of developing new portable software described in the previous subsection. The combination of scenarios for developing new software and porting existing software shows the need for iterative development of the process and the embedded evolutionary software life-cycle.

3.3 Building adaptive software for changing technologies

Clearly, it is not possible to anticipate every possible change in technology factors, such as transaction models or security support. However, it is often known in advance that some areas of a technology will change or are immature. In addition, companies have sometimes policies to not allow the use of a particular technology. The portability checker can be used to identify model elements that use such functionalities and can lead to potential problems. In contrast to the scenarios described above where the portability checker uses rules that are solely based on generically available UML profiles for different target platforms, in this situation we must modify these profiles so that they describe which model elements do not conform with a company's policies.

3.4 Using different MDA tools together

Working with just one MDA tool and its supported set of target platforms will not introduce new difficulties except as mentioned above. However, domain specific MDA tools and models will arrive. MOF based models can be incorporated in other MDA tools relatively easily with its MOF/XMI representation. However, the integration of code generators is more difficult. It is possible to use multiple MDA tools side-by-side. But there are some transitions in the models that are critical for interworking in the models. The portability checker can also be used to identify those transition elements that will need manual implementation work.

3.5 Integration of legacy systems

Very few systems are built to work in isolation. Many systems have to build upon already existing and running legacy systems that have to be integrated into the new system. Often there is no UML compliant model for these legacy systems and hence the integration has to be done manually bypassing an MDA tool. The portability checker can be extended with rules to verify how which interface elements of such a legacy system can be integrated automatically with MDA methods and which elements have to be hand-coded.

3.6 Reimplementation of an existing system

It is not a rare scenario that an existing system built without using MDA tools needs to be ported. The models for this system do not exist or are not usable as input for MDA tools. The portability checker cannot be applied directly on the binaries of this application. However, for certain platforms it is relatively straightforward to build a sketchy model of such a system by using the reflection methods of these platforms. This model is hardly complete but shows the coarse interrelationships of classes or components. Since this model is incomplete it cannot immediately be used as input for MDA tools. Interestingly, the portability checker can use such a model to

provide information on how difficult a reimplementation for a different target platform would be.

4 Example

In this section we present a simple example that elaborates the usefulness of the portability checker. It uses only the most simple constraint possible between metamodels: existence of similar concepts.

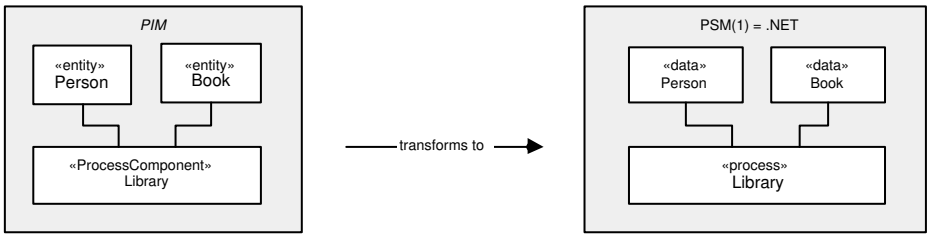


Fig. 1. PIM to .NET

Figure 1 shows a (narrowed) architecture for the administrative software of a library. The PIM on the left side makes use of extended UML constructs from OMG's Enterprise Collaboration Architecture (ECA) with its UML profiles for Component Collaboration Architecture (CCA) and UML profile for Entities. These profiles add UML stereotypes that represent entity objects such as persons or books and process objects such as the library itself. Although a UML profile for .NET has yet to be defined it is rather easy to automatically generate a .NET model and the source code from the available PIM.

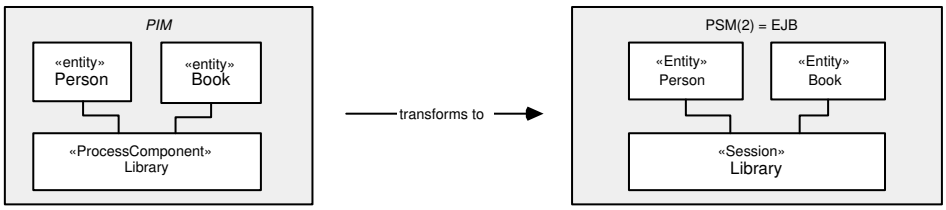


Fig. 2. PIM to EJB

The same application is later ported to Enterprise JavaBeans (see figure 2). Again this is straight-forward since there is no problem in finding a mapping of the PIM to a UML profile for EJB that maps entities to entity beans and process components to session beans. Both platforms are based on the same PIM.

A useful addition for this application is a notification mechanism that informs persons interested in a particular book that this book has been returned to the library. One way to design this is the use of an event fired by the returned book to the interested person. Events are also proposed in the ECA. The use of this event is shown in figure 3. The generation of the code

for .NET can be done easily because our .NET mapping is not constrained by any UML profile and .NET Remoting itself supports events even across remote boundaries.

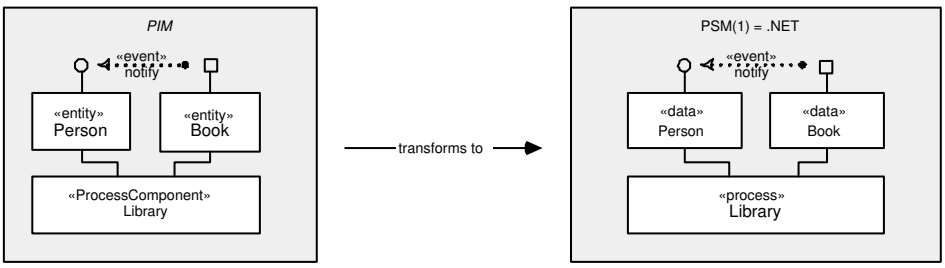


Fig. 3. PIM to .NET

However, what happens when the EJB application has to be updated to support the notification? The EJB UML profile does not support events. Furthermore enterprise beans do not currently support built-in callbacks. The portability checker knows that the EJB metamodel and the associated code generator do not support events and would return an appropriate warning.

5 Related Work

Different research areas are connected with our portability checker. When generating the output code an error can be generated when some inconsistency appears. Interestingly, none of the OMG specifications contain guidelines on what to do when a PIM to PSM mapping is incomplete or how to check this. Even the proposals for the upcoming OMG transformation standard for Queries, Views, and Transformations [3] mention the reaction to be an implementation issue.

In [1] the authors present a consistency-preserving approach for the evolution of UML models in real-time environments. They map UML-RT models to CSPs and investigate how changes of the model change the consistency of protocols or the preservation of deadlock freedom. Our work in contrast involves how the transition to a new and different target platform can be accomplished.

A framework for expressing transformations based on UML is presented in [7]. It supports an algorithm that checks if two UML models are refactorings of each other.

Since MOF and UML based models can be exported mechanically to XMI representation the problems we want to address can be considered as validating consistency of XML files with respect to constraints between these files. The xlinkit framework [5] supports checking the consistency of heterogeneous documents that are either already represented in XML or can be converted to XML. The constraints between these documents are described with a language based on first order logic. It has been restricted to be decidable in polynomial time [5]. The difference of xlinkit in contrast to other consistency checkers is

that it takes into consideration the cause and location why an inconsistency exists. In the future we expect to use xlinkit to support checking constraints between PIM and PSM at a very fine-grained level.

6 Conclusions

In this position paper, we have identified some practical software evolution issues that arise with MDA-based tools and transformations. We have proposed a tool, which we call a portability checker, to help address some of these problems. The portability checker relies on MOF/UML models and UML profiles to detect mismatches between source models and target platforms. With the help of the portability checker, designers and developers can design portable systems from scratch or port existing software to new platforms. The role of the portability checker is to automatically detect incompatibilities between features used in the model and those supported in the platform.

We are currently implementing the portability checker and plan to support different middleware platforms such as EJB, CORBA, and .NET. It is possible to generalize the notion of portability used here, which refers to compatibility with an execution platform and therefore is applicable to portability, to other areas and detect other incompatibilities. That is the subject of our current work.

7 Bibliographical references

References

[1] Engels, G., R. Heckel, J. M. Küster and L. Groenewegen, *Consistency-preserving model evolution through transformations*, in: *Proceedings of UML 2002* (2002), pp. 212–227.

[2] Frankel, D. S., "Model Driven Architecture: Applying MDA to Enterprise Computing," Wiley Publishing, 2003.

[3] Gardner, T., C. Giffin, J. Koehler and R. Hauser, *A review of omg mof 2.0 query / views / transformation submissions and recommendations towards the final standard* (2003).

[4] Mellor, S. J. and M. J. Balcer, "Executable UML: A Foundation for Model-Driven Architecture," Addison-Wesley, 2002.

[5] Nentwich, C., W. Emmerich, A. Finkelstein and E. Ellmer, *Flexible consistency checking*, ACM Trans. Softw. Eng. Methodol. **12** (2003), pp. 28–63.

[6] Soley, R., "Model Driven Architecture - White Paper," OMG (2000).

[7] Whittle, J., *Transformations and software modeling languages: Automating transformations in uml*, in: *Proceedings of UML 2002 (LNCS 2460)* (2002), pp. 227–242.

Proof Transformation via Interpretation Functions: Results, Problems and Applications

Piotr Kosiuczenko

Department of Computer Science
University of Leicester
Leicester, UK

Abstract

Change is a constant factor in Software Engineering process. Redesign of a class structure requires transformation of the corresponding OCL constraints. In a previous paper we have shown how to use, what we call, interpretation functions for transformation of constraints. In this paper we discuss recently obtained results concerning proof transformations via such functions. In particular we detail the fact that they preserve proofs in equational logic, as well as proofs in other logical systems like propositional logic with modus ponens or proofs using resolution rule. Those results have direct applications to redesign of UML State Machines and Sequence Diagrams. If states in a State Machine are interpreted by State Invariants, then the topological relations between its states can be interpreted as logical relations between the corresponding formulas. Preservation of the consequence relation can bee seen as preservation of the topology of State Machines. We indicate also an unsolved problem and discuss the mining of its positive solution.

Key words: UML, Redesign, Proof Transformation, Constraint Transformation, State Machines, Sequence Diagrams, Formal Methods.

1 Introduction

Unified Modelling Language provides textual and diagrammatic means for system specification (cf. [13]). Systems and their real-world environments are modelled using abstractions such as Classes Diagrams, State Machines or Sequence Diagrams. There are different software engineering processes which can be applied. In the old fashioned Waterfall Model, one has to begin with a correct requirement specification, make refinements to obtain the design and then implement the design specification. These steps can be adequately

[1] This research was partialy funded by the EU project Leg2Net and EU project AGILE.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

described using the notion of refinement (cf. e.g. [7] and [8]). This works correctly, if the requirements do not change and the software developers have a clear idea regarding how to proceed. In practice however, a specification constantly changes due to a number of factors including changed or new client requirements, new technology enablers and so on. In such a case extensive manual reengineering of system specification and design is needed. Today's software engineering processes embrace change as being a constant factor. In this case, requirements tracing is much harder to achieve. The notion of refinement with its monotonicity assumption can barely model such changes. For example, if an interface or class signature changes, a formula or a constraint which described a property concerning classes implementing this interface may no longer make sense. No tool in the market allows one for an automatic transformation of constraints. Changes have to be made to the specification manually, which is very time consuming and error prone.

A number of approaches to redesign of UML class models exist already. The best known is the refactoring approach [1]. It provides simple patterns for code and class structure modification to extend and modify a system without altering its behavior. The interpretation function, used in abstract algebra [11], transforms a single operation into a complex term. Graph rewriting systems may be used to transform specifications (cf. e.g. [3]).

In a previous paper [7] (see also [5]) we have studied the redesign of UML State Invariants with OCL constraints, as well as the transformation and tracing of constraints. We have introduced a new notion of interpretation function for redesign of State Invariants which extends, in a natural way, the notion introduced in [11]; an interpretation function is a compositional function generated by a mapping satisfying conditions analogous to orthogonality in term rewriting systems. Our concept of redesign is more general than the concept of refinement since we do not assume that properties are only added or refined, but we allow for changing them in an arbitrary way. For example a number of design level classes might be restructured or a specification level class might be split into several design level classes. Properties, which have to be preserved, may concern dependencies between classes, associations, attributes or generalization relationships. They are expressed by OCL formulas and transformed. Our approach is motivated by the notion of abstraction as it is used in UML [13]. Interestingly, our approach allows us for not only an automatic constraint transformation but also for an automatic proof transformation. In the technical report [6], we have shown that several kinds of entailment relations are preserved by interpretation functions; in particular such functions preserve equational proofs, proofs using propositional tautologies, resolution rule and induction. This allows one to save the clerical work of redoing proofs after transformation of a Class Diagram.

In this paper we present briefly the results contained in the technical report [5] and discuss their applications to redesign of State Machines and Sequence Diagrams. In section 2, we present briefly the idea of interpretation functions and the acompanying results. In section 3, we show how this idea and results can be applied to redesign of State Machines and Sequence Diagrams. In section 4, we conclude this paper and list some open problems.

2 Interpretation Functions

Interpretation functions proved to be very useful as a vehicle for an automatic transformation of OCL constraints when changes to Class Diagrams are performed [6] (see also [4]). For example, if an attribute a of type Integer is replaced by a path $b.x$, where b is an association pointing to another class and x is an attribute of that class, then every OCL constraint or State Invariant containing a has to be modified.

This kind of modifications can be performed using interpretation functions, i.e. partial functions generated by mappings satisfying conditions analogous to orthogonal term rewriting systems (cf. e.g. [12]). A domain of such a mapping satisfies conditions valid for all domains of orthogonal term rewriting systems; i.e. all terms in the domain are linear and non-overlapping.

Interpretation functions have several useful properties. They allow us to transform OCL specifications. The idea is that the designer or implementer who changes a class structure maps modified Model Elements on the target Model Elements, the transformation of the corresponding constraints being accomplished automatically. Such functions preserve equational proofs, proofs using propositional tautologies, resolution rule and proofs by induction [5]. This allows one to save the clerical work of redoing proofs of this kind after transformation of Class Diagrams.

3 Applications

Interestingly, results concerning preservation of consequence relation have also implications for redesign of other UML diagrams. In the following three subsections we consider application of our concepts to State Machines, Sequence Diagrams, and a simple example showing how the concepts work.

3.1 State Machines

One of the most useful kind of UML diagrams are the so called State Machines [13]. A State Machine is composed of a number of states connected by edges corresponding to transitions. States can be structured; one state can contain several other states called substates. In UML, "a state is a condition during the life of an object or an interaction during which it satisfies some condition, performs some action, or waits for some event" [13]. Consequently States

in a State Machine correspond to formulas; in the first case the formulas are called State Invariants. Such formulas may describe values of object attributes and inter-relationship between different objects; they can be expressed for example in OCL. The topological relation between states in a State Machine can be interpreted as logical relations between the corresponding formulas (cf e.g. [9]). In this case, preservation of the entailment relation can be seen as preservation of the topology of State Machines. It is natural to assume that the invariant corresponding to a substate implies the invariant corresponding to its superstate, since the invariant corresponding to the substate should be more restrictive. This condition (we call it state monotonicity) can be formally expressed as follows:
For every two states s_1 and s_2, s_1 is a substate of s_2 if and only if the formula corresponding to s_1 implies the formula corresponding to s_2.

On the other hand, one can be interested, if the states of a State Machine cover all possibilities. In the case of a State Machines describing the behavior of an object, this means that for every combination of the objects attributes, there is a state covering this combination. When states are interpreted by invariants, this covering property can be equivalently expressed by the requirement that for every combination of attributes, there is a State Invariant describing this combination. Formally, let F_i for $i = 1, ..., n$, be formulas corresponding to all states of a State Machine M; the states of M cover all possibilities if and only if the formula $F_1 \vee ... \vee F_n$ is a tautology. Equivalently, $F_1 \vee ... \vee F_n$ can be proved without using domain specific formulas.

A stronger property (we call it exhaustiveness) says that for every, so called, or-state all its substates cover all possibilities. Formally, let s be an or-state, let F be the corresponding State Invariant, let s_1,..., s_n be all substates of s and let $F_1, ..., F_n$ be the corresponding invariants. F is exhaustive if and only if F is logically equivalent to the disjunction $F_1 \vee ... \vee F_n$. We say that states in a State Machine are non-overlapping, if for every two different substates s_1 and s_2 of an or-state s, the conjunction of the corresponding invariants $F_1 \wedge F_2$ is logically false.

There are several other useful properties of this kind which can be expressed by logical relations between formulas. For example that all reachable states are defined by non-contradictory formulas or that disjunctions of formulas corresponding to orthogonal states are equivalent to the superstate invariant (a condition analogous to exhaustiveness).

As explained above, transformation of a Class Diagram requires the transformation of the corresponding constrains (for example OCL constraints). Consequently, if the states of a state Machine are described by formulas, those formulas may need to be changed. If implication in first order logic is preserved by interpretation functions, then all properties described above are preserved by such functions. The results presented in [5] show, that interpretation functions preserve proofs using equational reasoning, resolution rule, propositional

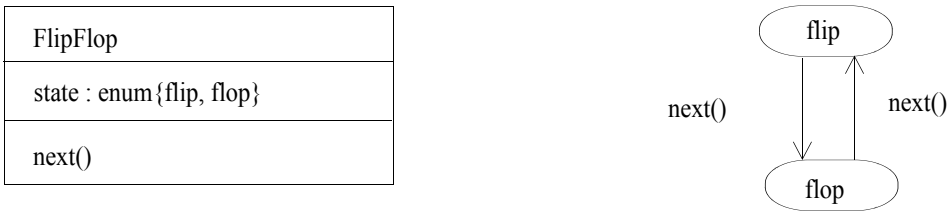
tautologies and induction. Consequently at the moment we can say that if the above mentioned properties are proved using those kinds of reasoning, then they are preserved by interpretation functions. If for example, there is a proof of the state-monotonicity property using above mentioned ways of reasoning, then after transformation via an interpretation function this property remains valid after transformation.

3.2 Sequence Diagrams

UML 2.0 introduces conditions to Sequence Diagrams [14]. The previous section contains a list of State Machines properties which can be defined in logical terms. In a similar way one can define properties of Sequence Diagrams. For example, one can require that conditions in a Sequence Diagram, which is obtained using parallel composition, are non-contradictory or that Sequence Diagrams combined by alternative composition have exclusive pre-conditions. As in the case of State Machines, the preservation of logical consequences would imply that such properties are preserved by interpretation functions.

3.3 Example

In this subsection we apply our concepts to a concrete example. We consider a refactoring of a class diagram according to so called State Pattern [2]. First we show an implementation of a State Machine by enumeration types; then we redesign this implementation using the State Pattern. We refer the interested reader to [6] for the details of this pattern application. First, we implement states using an enumeration type:

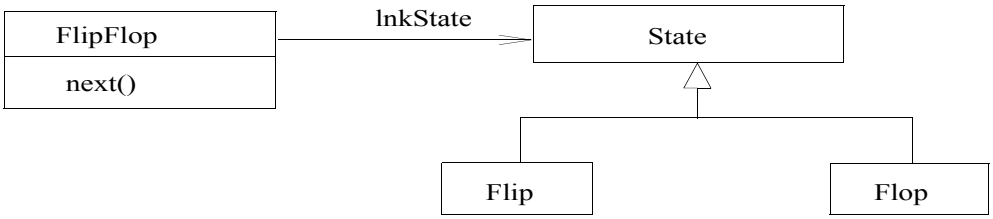


The figure above shows a state machine diagram for the class *FlipFlop* with two or-states. An object of this class can be either in the state *flip* or *flop*. There exists an operation *next*(), which transposes the states. We prove that

these states are non-overlapping by contradiction using propositional tautologies and equational reasoning:

The formula $self.state = \#flop \wedge self.state = \#flip$ implies the formula $\#flop = self.state \wedge self.state = \#flip$, thanks to the tautology $(a \Rightarrow b) \Rightarrow (c \wedge a \Rightarrow c \wedge b)$ and the symmetry rule for equations $x = y \Rightarrow y = x$. The formula $\#flop = self.state \wedge self.state = \#flip$ implies $\#flop = \#flip$, thanks to transitivity rule for equations: $x = y \wedge y = z \Rightarrow x = z$. The elements of enumeration type are different, therefore $\#flip \neq \#flop$. We apply the tautology $(p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow q)$ and the tautology $(p \Rightarrow q) \wedge \neg\, q \Rightarrow \neg\, p$ and obtain formula $\neg\, (self.state = \#flop \wedge self.state = \#flip)$. Consequently, the states *flip* and *flop* are non-overlapping.

The State Pattern application yields the following class diagram:



In the redesigned version, the states are implemented by objects instantiating classes *Flip* and *Flop*, which subclass the class State (see the figure above). We map the elements of the enumeration class on the corresponding classes; i.e. *flip* and *flop* are mapped to classes *Flip* and *Flop*, respectively. Moreover, the attribute *state* is mapped to the term *lnkState.oclType*. This mapping, defined on the level of model elements, induces an interpretation function (cf. [6]).

We show that the property of non-overlappingness of the or-states is preserved by the interpretation function. The results proved in [5] guarantee, that the interpretation function preserves proofs like the one above. Indeed, the previous proof can be transformed; the transformed proof has the same form as the previous one:

The formula $self.lnkState.oclType = Flop \wedge self.lnkState.oclType = Flip$ implies the formula $Flop = self.lnkState.oclType \wedge self.lnkState.oclType = Flip$, thanks to the tautology $(a \Rightarrow b) \Rightarrow (c \wedge a \Rightarrow c \wedge b)$ and the symmetry axiom for equations: $x = y \Rightarrow y = x$. The formula $Flop =$

$self.lnkState.oclType \wedge self.lnkState.oclType = Flip$ implies the formula $Flop = Flip$, thanks to transitivity rule for equations: $x = y \wedge y = z \Rightarrow x = z$. The classes *Flip* and *Flop* are different. We apply the tautology $(p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow q)$ and the tautology $(p \Rightarrow q) \wedge \neg\, q \Rightarrow \neg\, p$ and obtain formula $\neg\, (self.lnkState.oclType = Flop \wedge self.lnkState.oclType = Flip)$. Therefore in this case, the states *flip* and *flop* exclude each other and consequently the application of State Pattern preserves the property of non-overlappingness.

4 Concluding Remarks and Future Work

In this paper we show that results presented in [6] and in [5] have direct applications to transformation of UML diagrams, in particular to State Machines and Sequence Diagrams. We show that logical relations between State Invariants in a State Machine are preserved, if they can be proved using certain kinds of proofs.

There are still several open questions. In particular, it is not known, if interpretation functions preserve entailment relation of first order logic. This question may seem purely theoretical, but a positive answer would have very interesting implications for State Machines and Sequence diagrams; it would mean, for example, that interpretation functions preserve topology of State Machines. Even if the answer is negative, the results obtained so far prove to be useful.

In the future, we are going to further study the properties of interpretation functions in logical terms. We are going to investigate, if the entailment relation of first order logic is preserved by interpretation functions. In first order logic there is an equivalence between semantic and syntactic consequence. Our results obtained up to now focus mainly on the syntax. Institutions approach the problem of transformation from the model theoretic point of view (cf. e.g. [10]). We are going to study the relation of interpretation functions to institutions. We plan also to implement a tool supporting class redesign, transformation and tracing of model elements. Such a tool will be very helpful since a purely manual transformation of complex OCL constraints is very laborious and error prone.

References

[1] Fowler, M., *Refactoring: Improving the Design of Existing Code*, Reading, Mass., Addison-Wesley, 2000.

[2] Gamma, E., Helm, R., Johnson, R., and J. Vlissides, *Design Patterns*, Addison-Wesley, Reading, 1995.

[3] Grosse-Rhode, M., and Presicce, F., and M. Simeoni, *Formal software specification with refinements and modules of typed graph transformation systems*, Journal of Computer and System Sciences. **64(2)** 2002.

[4] Kosiuczenko, P., *Formal Redesign of UML Class Diagrams*, in (Evans, A., France, R., Moreira, A., Rumpe, B. Eds): Proc. of pUML Workshop on Practical UML Based Rigorous Development Methods, Toronto. GI-Edition, Lecture Notes in Informatics, 2001.

[5] Kosiuczenko, P., *Proof Transformation via Interpretation Functions*, Technical Report nr. 2004/27, University of Leicester, 2004, 16 pages, http://www.cs.le.ac.uk/ pk82/Theory1.1.pdf.

[6] Kosiuczenko, P., *Redesign of UML Class Diagrams: a formal Approach*, submitted for publication, 2004.

[7] Lano, K., *Formal object oriented development*, Springer, Berlin, 1995.

[8] Lechner, U., *Object-Oriented Specification of Distributed Systems*, Dissertation, Universitaet Passau, Fakulaett fuer Mathematik und Informatik, 1997.

[9] Luettgen, G., and M. Mendler, *Statecharts: From Visual Syntax to Model-Theoretic Semantics*, in proc. of Wirtschaft und Wissenschaft in der Network Economy, Tagungsband der GI/OCG-Jahrestagung, K. Bauknecht, W. Brauer, Th. Mck (eds.), Viena, 25.09.2001..

[10] Tarlecki, A., *Institution: An Abstract Framework for Formal Specifications*, In (Astesiano, E., Kreowski, H. -J., Krieg-Brckner, B. Eds.): Algebraic Foundations of System Specification, Springer 1999.

[11] Taylor, W., *Characterizing Malcev conditions*, Algebra Universalis, **3**, Springer, Berlin, 1973, 351–397.

[12] Terese et. al., *Term rewriting systems*, Cambridge University Press, 2003.

[13] OMG, *Unified Modeling Language Specification*, Version 1.5, 2003.

[14] OMG, *Unified Modeling Language Specification*, Version 2.0 (pending), 2004.

On the Evolution Complexity of Design Patterns

Tom Mens [1]

Software Engineering Lab, Université de Mons-Hainaut
B-7000 Mons, Belgium

Amnon H. Eden [2]

Department of Computer Science, University of Essex
and Center For Inquiry International

Abstract

Software co-evolution can be characterised as a way to "adjust" any given software implementation to a change ("shift") in the software requirements. In this paper, we propose a formal definition of evolution complexity to precisely quantify the cost of adjusting a particular implementation to a change ("shift") in the requirements. As a validation, we show that this definition formalises intuition about the evolvability of design patterns.

Key words: software evolution, design pattern, complexity

1 Introduction

According to Lehman's first law of software evolution, "*An E-type program that is used must be continually adapted else it becomes progressively less satisfactory.*" [10]. Despite growing awareness of this law, evolution of industrial quality software systems is notoriously expensive. It is therefore paramount to investigate the flexibility or evolvability of software and to find ways to quantify it.

Claims about the evolvability of design patterns, architectural styles and object-oriented programs have appeared in numerous publications. Most authors, however, stop short from quantifying the benefits gained by using a particular implementation policy or qualifying the claim by the class of "shifts" (i.e., changes in the software requirements) it best allows.

[1] Email:tom.mens@umh.ac.be
[2] Email:eden@essex.ac.uk

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Experience shows that flexibility is relative to the change. Every manufactured product is designed to accommodate to a specific class of changes, which makes it flexible towards these changes but inflexible towards others. Locomotives, for example, are very flexible with relation to the type of cars they can pull but they can hardly be adapted to tracks of a different size.

The same applies to software: Every implementation policy (e.g., architectural style [14], design pattern [5]) is flexible towards the class of changes it was designed for. For example, a program designed using Layered Architecture style, such as the Unix operating system, adapts easily to changes in the uppermost layer (the *application layer*) but changes to the lowermost layer (the *kernel*) are much more difficult to implement. As another example, the Visitor design pattern "*makes adding new operations easy*" while "*adding new concrete element classes is hard*" [5] (pp. 335–336).

There is a common misconception in the software engineering community, however, that flexibility and evolvability are absolute qualities. Consider for example the following standardised definitions:

Flexibility The ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed [6]

Extendability The ease with which a system or component can be modified to increase its storage or functional capacity [6]

Changeability The capability of the software product to enable a specified modification to be implemented [7]

These definitions fail to observe that flexibility of a program is relative to a particular class of changes. As a notable exception, Gamma *et al.* were more precise in characterising the quality of flexibility [5] : "*Each design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a* ***particular kind of change***."

In this paper we examine the relationship between *shifts*, i.e., changes in the requirements, and *adjustments*, i.e., the respective changes in the implementation. We offer a precise definition of *evolution complexity* and propose various metrics to approximate its cost. To illustrate its usefulness, we compute the evolution complexity of adjusting a selected number of design patterns to specific shifts and prove informal statements about how difficult is each. For example, we prove that the "*Visitor pattern makes adding new operations easy*" by showing that the evolution complexity of adding an operation is constant, and that "*Adding new concrete element classes is hard*" by showing that the evolution complexity of adding a new concrete element is linear in the number of visitors.

Furthermore, we use evolution complexity to show that whether one implementation policy is more evolvable than another depends on the class of changes in question. For example, given the requirements for representing a deterministic finite-state automaton, we demonstrate that a procedural imple-

mentation is more evolvable towards shifts in the alphabet, while an object-oriented implementation (using the State pattern) is more evolvable towards shifts in the set of states.

To summarise, the intended contributions of this paper are: (a) To formalise and prove the intuition behind flexibility and evolvability of specific implementation policies, in particular of design patterns; (b) To provide means for choosing a particular implementation policy; (c) To provide means for quantifying the cost of implementing specific changes;

2 Setting the scene

In this section, we clarify the terminology that will be used in the remainder of the paper. These definitions are summarized in Table 1 and illustrated in Figure 1.

Table 1
Key to notation

requirements	$r_{dfsa}, r_{gui}, \ldots$
implementations	$i_{before}, i_{after}, \ldots$
problem domain	$\mathcal{D} = \{D_1, \ldots, D_n\}$
shift	$\sigma = \sigma_\delta(r, U, u)$
adjustment	$\alpha = (i, i')$
co-evolution step	$\epsilon = (\sigma, \alpha)$

Requirements. A well-defined specification of the program's expected behaviour, expressed in terms of the problem domain $\mathcal{D}$. For example:

r_{dfsa} := *Implement a deterministic finite state automaton with states S and alphabet L.*

In this case, the problem domain $\mathcal{D}_{dfsa} = \{S, L\}$.

r_{gui} := *Implement a GUI to represent and instantiate a family of widgets (e.g., button, window and menu) in a specified set of operating systems (e.g., Windows and MacOS).*

In this case, the problem domain $\mathcal{D}_{gui} = \{W, O\}$, where W = { button, window, menu } and O = { Windows, MacOS }.

Implementation. The actual program that is the subject of the evolution effort. Each implementation is designed to satisfy a specific set of requirements and must be adjustable to changes therein. In this paper, we will use the term *implementation policies* to reflect that fact that we abstract away from language-specific details.

Shift. A specific change to a given set of requirements. More specifically, a shift may add entities to, or remove entities from, the sets contained in the

problem domain $\mathcal{D}$. A shift is represented as a function $\sigma_\delta(r, D, d)$ where $D \in \mathcal{D}$, and d is added to D if $\delta =$"+", while d is removed from D if $\delta =$"−".

For example, the shift $\sigma_+(r_{dfsa}, L, l)$ adds a letter l to the alphabet L in r_{dfsa}.

Adjustment. A specific change to a given implementation, triggered by a shift in the requirements. As shown in Figure 1, each implementation needs to be "adjusted" in order to satisfy the changed requirements. An adjustment is represented as a pair $\alpha = (i, i')$ where i is the old implementation and i' is the new implementation.

Co-evolution step. A pair $\epsilon = (\sigma, \alpha)$ consisting of a shift σ transforming r into r' and an adjustment $\alpha = (i, i')$, such that implementation i satisfies requirements r and implementation i' satisfies requirements r'.

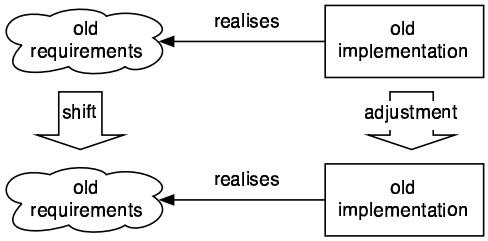


Fig. 1. A co-evolution step.

3 Evolution complexity

In this section, we use the previously introduced vocabulary to define a notion of *evolution complexity*. With this definition we attempt to quantify the cost of a co-evolution step, i.e., the effort that is required to *adjust* a specific implementation to a particular *shift*.

In order to define evolution complexity, we draw an analogy with the notion of *computational complexity*. "The theory of computational complexity is concerned with estimating the resources a computer needs to solve a problem." [16] In analogy, evolution complexity is concerned with estimating the resources required to "evolve" an implementation. Using metaprogramming as a central metaphor, we suggest that evolution complexity, i.e., the complexity of the process of adjusting the implementation to changed requirements, can be approximated by calculating the computational complexity of a metaprogramming algorithm that actually makes these adjustments to the software implementation. This leads us to the most general formulation of evolution complexity:

Definition 3.1 The complexity $\mathcal{C}(\epsilon)$ of a co-evolution step ϵ is the complexity of the meta-programming algorithm that implements it.

Note that there are as many meta-programming algorithms (for implementing the adjustment) as there are manual ways for implementing them. We seek to approximate the manual (adjustment) process by measuring the respective (hypothetical) metaprogramming process.

Since $\epsilon = (\sigma, \alpha)$ is a pair, this definition implies that the actual effort required to "evolve" a program correlates primarily with two factors:

(i) The **shift** σ, i.e., the distance between the old and the new requirements;

(ii) The **adjustment** α, i.e., the distance between the implementation before and after the change.

The obvious question is: How can we measure these distances? In the following section, we illustrate some of the ways to approximate evolution complexity.

4 Case studies

In this section, we use evolution complexity to quantify the evolvability of different implementation policies. We do this in two different ways. To compare the evolvability of different implementations (e.g., procedural versus object-oriented implementation), we fix the shift and calculate the evolution complexity for each implementation. To compare the difficulty of implementing different shifts, we fix the implementation and calculate the evolution complexity for each shift.

The presentation of each case study will be structured in the same way: *Example*; *Requirements*; *Shifts*; *Implementation*; *Metric*; *Analysis*; *Discussion.*

4.1 Case study 1: Visitor

The Visitor design pattern [5] can be used to "*Represent an operation to be performed on the elements of an object structure.*" The class of shifts that the Visitor supports is declared from the outset: "*Visitor lets you define a new operation without changing the classes of the elements on which it operates.*" Further on in the chapter, the authors are more explicit: "*Visitor makes adding new operations easy*" and "*Adding new concrete element classes is hard*". In this case study, we prove these statements and quantify how easy or hard is each one of these co-evolution steps.

Example

Gamma *et. al* describe the representation of abstract syntax trees as object structure and the collection of operations that a compiler performs on each element in the tree. Figure 2 illustrates this example.

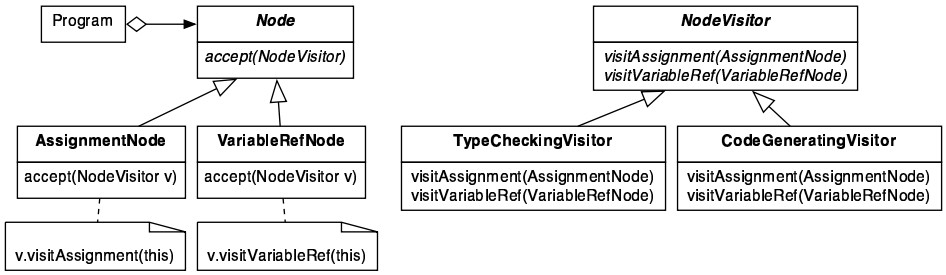


Fig. 2. Example for the Visitor pattern.

Requirements

r_{visit} := *Represent a set of operations* O *that need to be performed on a family of elements* E.

The problem domain is $\mathcal{D}_{visit} = \{O, E\}$

Shifts

In this case study we will consider the following two shifts to r_{visit}:

$\sigma_+(r_{visit}, O, op)$ = *add operation* op *to* O

$\sigma_+(r_{visit}, E, e)$ = *add element* e *to* E

Implementation

i_{visit} := The implementation policy described by the Visitor pattern as illustrated in Figure 2.

By this policy, two class hierarchies, `NodeVisitor` and `Node` are used to represent the set of operations O and the family of elements E, respectively. Each operation $op \in O$ is represented as a class in the `NodeVisitor` hierarchy, and each element $e \in E$ is represented as a class in the `Node` hierarchy.

Metric

Below we definite a simple metric that calculates the complexity of a co-evolution step by counting the number of classes affected by it.

Definition 4.1 Let (σ, α) be a co-evolution step such that $\alpha = (i, i')$. Let Δ be the symmetric set difference, i.e., $A \Delta B = (A \backslash B) \cup (B \backslash A)$. Let $Classes(i)$ = the set of classes in i .

$\mathcal{C}^1_{Classes}(\sigma, \alpha) := |Classes(i) \Delta Classes(i')|$

In class-based languages such as C++, Smalltalk and Java, $Classes(i)$ yields the set of classes defined in the program. For example,
$Classes(i_{visit})$ = {`Node`, `AssignmentNode`, `VariableRefNode`, `NodeVisitor`, `TypeCheckingVisitor`, `CodeGeneratingVisitor`}

Analysis

Gamma *et. al* recognized the class of shifts that the visitor accommodates to. We will use the above metric to prove some of their claims about the flexibility

of the Visitor pattern:

- "*Adding new concrete element classes is hard.*" ([5], p.335) To prove this statement, consider the shift $\sigma_+(r_{visit}, E, e)$. In order to adjust our implementation i_{visit} to this shift, we need to add a method to every class in the visitors hierarchy O. Thus, the number of implementation entities affected by this shift equals the number of operations:

$$\mathcal{C}^1_{Classes}(\sigma_+(r_{visit}, E, e), i_{visit}) = |O|$$

  Read: *The class-level evolution complexity of adding an element to the Visitor pattern is linear in the number of operations.*

- "*Visitor makes adding new operations easy.*" ([5], p.336) To prove this statement, consider the shift $\sigma_+(r_{visit}, O, op)$. In order to adjust our implementation i_{visit} to this shift, we need to add a new class to the visitors hierarchy. Thus, the number of implementation entities affected by this shift is 1:

$$\mathcal{C}^1_{Classes}(\sigma_+(r_{visit}, O, op), i_{visit}) = 1$$

  Read: *The class-level evolution complexity of adding an operation to the Visitor pattern is constant.*

The results of this case study are summarized in Table 2.

$\mathcal{C}^1_{Classes}$	$\sigma_+(r_{visit}, E, e)$	$\sigma_+(r_{visit}, O, op)$
i_{visit}	$\|O\|$	1

Table 2
Class-level evolution complexity for shifts of r_{visit}

Discussion
Our analysis proves the intuitions of Gamma *et al.* about the Visitor pattern, but it also goes beyond that: The evolution complexity metric quantifies the effort required to add a new element to the elements hierarchy. Specifically, it indicates that effort required to add a new operation is proportional to the number of operations.

4.2 Case study 2: Abstract Factory

The Abstract Factory pattern can be used to "*Provide an interface for creating families of related or dependent objects without specifying their concrete classes.*" [5] Below, we show that Abstract Factory is only flexible towards specific shifts, and that the adjustments necessary can also be very expensive.

Example
A typical example is an object-oriented graphical user interface (GUI) for creating graphical "widgets", such as windows, buttons and menus. Each win-

dowing environment offers a variation on each one of these widgets. Figure 3 illustrates the "families" of widgets for three windowing environments.

A client in cross-platform implementation that needs a new button, for example, must decide which variation of button is appropriate. The obvious way to do this would be to use complex conditional statements that determine the appropriate variation. The alternative, offered by the Abstract Factory design pattern, is to offer each such client a uniform interface for generating each widget, and to delegate the decision which version is appropriate to a "concrete factory" object. The dashed lines in Figure 3 illustrate the "create" relation between factory method, concrete factories and the products hierarchy.

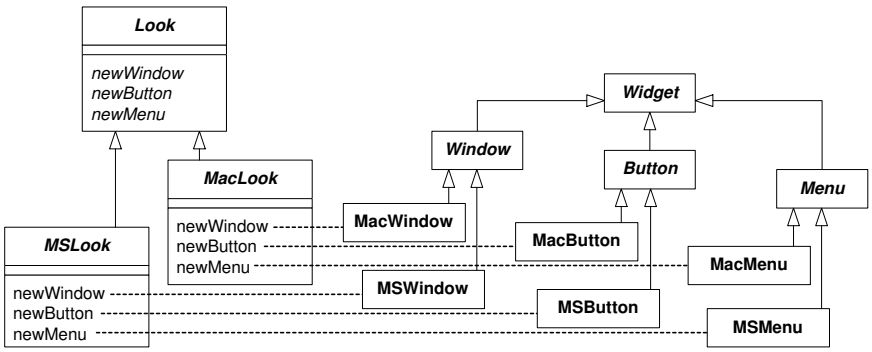


Fig. 3. Example for *Abstract Factory*.

Requirements

$r_{conf/prod}$:= *Given a set of clients* K*, a family of configurations* $C = \{c_1, ...c_n\}$*, and a family of products* $P = \{p_1, ...p_m\}$*. Every client* k *in* K *needs a new instance* p *in* P *depending on the current configuration* c *in* C*.*
The problem domain $\mathcal{D}_{conf/prod} = \{K, C, P\}$.

Shifts

In this case study we will consider the following two shifts to $r_{conf/prod}$:

$\sigma_+(r_{conf/prod}, C, c)$:= *add configuration* c *to* C
$\sigma_-(r_{conf/prod}, P, p)$:= *remove product* p *from* P

Implementation policies

We will compare two implementation policies to $r_{conf/prod}$:

$i_{abs-factory}$:= use the Abstract Factory design pattern ([5], page 87). The solution this pattern dictates consists of two class hierarchies, *factories* and *products*. Each class in the *factories* hierarchy ("concrete factory") is responsible for creating products of a specific configuration $c \in C$. For this purpose, each concrete factory defines a method ("factory method") for each product $p \in P$. Overall, there are $|C|$ concrete products with $|P|$ methods each in this implementation.

i_{cond} := use conditional style. Each client that needs a specific product implements a switch statement that has a conditional branch for each configuration $c \in C$.

In both implementations, we assume that there are $|K|$ separate clients (i.e., $|K|$ classes in a class-based language) such that each client needs a new instance of one or more products.

Analysis

An example of shift $\sigma_-(r_{conf/prod}, P, p)$ would be to remove the product `Menu` in Figure 3. This requires us to remove all concrete subclasses of `Menu`, as well as the methods `newMenu` that are implemented in each concrete factory.

Table 3
Class-level evolution complexity for shifts of $r_{conf/prod}$

$\mathcal{C}^1_{Classes}$	$\sigma_+(r_{conf/prod}, C, c)$	$\sigma_-(r_{conf/prod}, P, p)$
$i_{abs-factory}$	$\|P\|$	$\|C\| + \|P\|$
i_{cond}	$\|K\|$	$\|K\|$

The effect of implementation policy $i_{abs-factory}$ on the evolution complexity is described informally in [5], pages 89 and 90: "*It makes exchanging product families easy*"; "*supporting new kinds of products is difficult*". Again, we will quantify exactly how easy or difficult it is.

As shown in Table 3, the evolution complexity at class level of $i_{abs-factory}$ is linear for both shifts. For shift $\sigma_+(r_{conf/prod}, C, c)$, it is *linear in the number of products*, since we need to add a new concrete product class for each possible product, to specify how each product needs to be addressed by the new configuration. For shift $\sigma_-(r_{conf/prod}, P, p)$, it is *linear in the number of configurations* ***and*** *products*, since we need to remove all concrete product classes for the considered product, and we need to remove a corresponding method in each of the configuration classes.

For implementation i_{cond}, the evolution complexity is *linear in the number of clients* for both shifts.

Discussion

The results presented in Table 3 demonstrate that the decision to use the Abstract Factory design pattern is not straightforward. If the number of clients is very small then the overhead in using the pattern is not justified. Similarly, it shows that removing a product can be a labour-intensive task even if the pattern is used.

Obviously, the evolution complexity is not the only criterion that should be used for preferring a particular implementation policy over another. For example, the *Abstract Factory* has a number of other important advantages that are

not measured by our evolution complexity measure: "*it isolates clients from implementation classes*"; "*it promotes consistency among products*". Thus, evolution complexity only captures one of many concerns that guide designers in the choice of a particular implementation policy.

4.3 Case study 3: Procedural vs. object-oriented implementation

Object-oriented programming is hailed for promoting flexibility. Experienced programmers, however, observe that flexibility is relative to a specific class of changes that our implementation must specifically be designed to accommodate. As our analysis of the Visitor pattern demonstrated, changes to the interface of the base class in a large inheritance class hierarchy can be very expensive to implement, but adding a new "leaf" class is usually very easy.

In this case study, we use evolution complexity to compare the flexibility of a procedural and an object-oriented implementation policy in the context of a specific problem: the representation of a finite-state automaton. Our analysis will demonstrate that an object-oriented implementation is more flexible towards changes in the set of states, while the procedural implementation is more flexible towards changes in the alphabet.

Example
Consider a digital clock with three display states *Display Hour*, *Display Seconds* and *Display Date* and two setting states *Set Hour* and *Set Date*. The clock accepts input from two buttons b_1 and b_2, which are used to change between states or to perform a specific action depending on the current state. The clock behaviour is modelled in Figure 4 as a deterministic finite state automaton with set of states S={*Display Hour*, *Display Seconds*, *Display Date*, *Set Hour*, *Set Date*} and an alphabet L={b_1,b_2}.

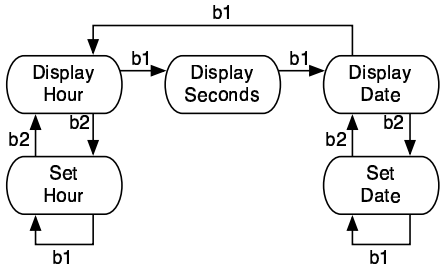


Fig. 4. State machine for a digital clock.

Requirements
r_{dfsa} := *Implement a deterministic finite state automaton (DFSA) with a set of states S and a set of letters L (i.e., an alphabet).*
The problem domain $\mathcal{D}_{dfsa} = \{S, L\}$.

Shifts

In this case study we will consider the following two shifts to r_{dfsa}:

$\sigma_+(r_{dfsa}, L, l) :=$ *add letter* l *to the alphabet* L

$\sigma_+(r_{dfsa}, S, s) :=$ *add state* s *to the set of states* S

Implementations

We will compare two implementation policies for the digital clock:

i_{state} = use the *State* design pattern ([5], p.305). This pattern dictates that each state $s \in S$ is represented by a separate class such that every state class defines a method for each $l \in L$. In all, there are $|S|$ classes with $|L|$ methods per class in this implementation. This implementation is illustrated below:

```
interface ClockState {
  void b1(); \\ button 1 pressed
  void b2(); \\ button 2 pressed
}
class DisplayHour implements ClockState {
  public void b1() {\* b1 pressed *\}
  public void b2() {\* b2 pressed *\}
}
class DisplaySecond implements ClockState {
  public void b1() {\* b1 pressed *\}
  public void b2() {\* b2 pressed *\}
}
class DisplayDate implements ClockState {
  public void b1() {\* b1 pressed *\}
  public void b2() {\* b2 pressed *\}
}
class SetHour implements ClockState {
  public void b1() {\* b1 pressed *\}
  public void b2() {\* b2 pressed *\}
}
class SetDate implements ClockState {
  public void b1() {\* b1 pressed *\}
  public void b2() {\* b2 pressed *\}
}
```

i_{cond} = use *conditional style*, ([5], p.307): "*An alternative is to use data values to define internal states and have context operations check the data explicitly. But then we'd have look-alike conditional or case statement scattered throughout the context's implementation.*" In this style of implementation, we define one class that contains a method for each letter $l \in L$ such that the body of each method consists of a switch statement that contains a conditional branch for each state $s \in S$. This implementation is given below:

```
enum states = {
  DisplayHour, DisplaySecond, DisplayDate,
```

```
  SetHour, SetDate} state;  // Current state

void b1() { \\ button 1 pressed
  switch (state) {
    case DisplayHour: \*...*\;
    case DisplaySecond: \*...*\;
    case DisplayDate: \*...*\;
    case SetHour: \*...*\;
    case SetDate: \*...*\; }
}
void b2() { \\ button 2 pressed
  switch (state) {
    case DisplayHour: \*...*\;
    case DisplaySecond: \*...*\;
    case DisplayDate: \*...*\;
    case SetHour: \*...*\;
    case SetDate: \*...*\; }
}
```

Metric

The class-level metric of Definition 4.1 is too coarse since it assumes that all software entities require approximately the same effort to be adjusted. Such an approximation is useful when the number of software entities is large or when there is insufficient detailed information about individual entities. For example, at the design level, before the implementation is complete, we may only have information about the classes and their variables, but not necessarily about their methods. When the implementation is complete, however, and when it is evident that some software entities are more difficult to adjust than others, we may replace the class-level metric by a generalized metric that measures the (software) evolution complexity of individual modules:

Definition 4.2 Let (σ, α) be a co-evolution step such that $\alpha = (i, i')$. Let Δ be the symmetric set difference. Let $Modules(i)$ = the set of modules in i. Let μ be a software complexity metric that is defined for all $x \in Modules(i) \cup Modules(i')$,

$$\mathcal{C}^{\mu}_{Modules}(\sigma, \alpha) := \sum_{x \in Modules(i) \Delta Modules(i')} \mu(x)$$

Let us show how the generalized metric can be used in specific cases:

$\mathcal{C}^{\mathbf{1}}_{\mathbf{Classes}}$ When $Modules = Classes$ and the software complexity $\mu(c) = 1$ for all $c \in Classes(i)$, the generalized metric is reduced to the class-level evolution complexity metric of Definition 4.1.

$\mathcal{C}^{\mathbf{1}}_{\mathbf{Methods}}$ When $Modules = Methods$ (the set of all methods in i) and the software complexity $\mu(m) = 1$ for all $m \in Methods(i)$, we have a method-level evolution complexity metric. This metric measures the complexity of

a co-evolution step by counting the number of methods affected by it. It treats all methods as equal with respect to their change effort.

$\mathcal{C}^{\mathbf{CC}}_{\mathbf{Methods}}$ When $Modules = Methods$ and $\mu = CC$ (*cyclomatic complexity*, [12]), the metric takes into account the cyclomatic complexity of each method such that adjustments applied to a "complicated" method (e.g., a method with many conditional statements) are more expensive than adjustments applied to a "simple" method.

Analysis

The effect of the two implementation policies i_{state} and i_{cond} on the evolution complexity is described informally in [5]. For example, pages 307 and 308 mention for i_{state}: "new states ... can be added easily"; "decentralizing the transition logic in this way makes it easy to modify or extend the logic". For i_{cond}, page 307 mentions: "Adding a new state ... complicates maintenance." Our formal framework lets us precisely quantify this intuition in terms of the entities of the problem domain $\mathcal{D}_{dfsa} = \{S, L\}$ that are affected by a particular shift, assuming a particular implementation policy.

$\mathcal{C}^{1}_{Modules}$	$\sigma_+(r_{dfsa}, L, l)$	$\sigma_+(r_{dfsa}, S, s)$
i_{state}	$\|S\|$	1
i_{cond}	1	$\|L\|$

Table 4
Class-level evolution complexity for shifts of r_{dfsa}

Analysing the evolution complexity at the class level (Table 4), we observe for shift $\sigma_+(r_{dfsa}, L, l)$ that implementation policy i_{state} is more difficult to evolve than implementation policy i_{cond}. Indeed, the *State* design pattern requires us to add a new method (corresponding to the new letter to be added) to each of the $|S|$ state classes. Hence, the evolution complexity for i_{state} is *linear in the number of states*, whereas it is constant for i_{cond}.

$\mathcal{C}^{CC}_{Methods}$	$\sigma_+(r_{dfsa}, S, s)$
i_{state}	$\|L\|$
i_{cond}	$\|S\| \times \|L\|$

Table 5
Method-level evolution complexity for shifts of r_{dfsa}

Focussing on the method level (Table 5) shows us two other important results. At this level of abstraction, the evolution complexity for shift $\sigma_+(r_{dfsa}, L, l)$ is the same for both implementation policies (linear in the number of states). Shift $\sigma_+(r_{dfsa}, S, s)$, on the other hand, is more difficult to

evolve for implementation policy i_{cond} than for i_{state} (complexity $|S \times L|$ and $|L|$, respectively). These results were obtained by using the cyclomatic complexity metric CC for each affected method. It reflects the intuition that an implementation policy with lots of conditional statements is more difficult to evolve than one with few conditionals.

Discussion
While Table 4 shows that *adding a state* is constant in the number of states, *removing a state* with implementation policy i_{state} is potentially linear in the number of states, since we need to modify the state transition function, which is embedded in, and dispersed over, all state classes. As an alternative implementation policy, we could opt for a variant of the *State* design pattern, where state transitions are implemented in the context. Needless to say, this implementation policy will yield different results for the evolution complexity when compared to i_{state} and i_{cond}.

5 Related work

Despite its importance, cost estimation in the context of software maintenance and software evolution remains relatively unexplored. Jørgensen [8] used several models to predict the effort of randomly selected software maintenance tasks. The size of individual maintenance tasks was measured in LOC. Sneed [15] proposed a number of ways to extend existing cost estimation methods to the estimation of maintenance costs. Ramil *et al.* [13] provided and validated six different models that predict software evolution effort as a function of software evolution metrics.

All of the above approaches rely make use of software metrics, and can rely on experimentally validated results of the software metrics community [3,4,12,17]. While our definition can also incorporate existing software complexity metrics (e.g., cyclomatic complexity) easily, an important distinction is that our approach goes beyond existing attempts to measure the "evolvability" of implementations, since we have shown that it is not possible to give an absolute measure of the evolvability of a particular program. Instead, the evolvability depends on the chosen implementation policy and on the changes in the requirements that are likely to be performed.

Perhaps a better (i.e., less absolute) way to measure the changeability of a software system relies on algorithms or measures that compute the impact of changes [11]. For example, Chaumun *et al.* [2] report on experimental results with a change-impact model for object-oriented systems.

Because the cost and complexity of software evolution may depend on the type of evolution activity, we also require a more objective and finer granularity recognition of types of software evolution activities. An attempt to make such an objective classification of evolution activities was carried out in [1].

6 Conclusion

In this paper we proposed a formal definition of evolution complexity to quantify the cost of adjusting a particular implementation policy to a change ("shift") in the requirements. As a case study, we used our formalism to formally validate the intuition that design patterns improve the evolvability of programs. We were able to determine precisely to which extent, and for which changes in the requirements, a particular design pattern is "more evolvable" than another implementation policy. Despite all this evidence, a more scientific empirical validation of our proposed evolution complexity metric remains to be done.

In general terms, our formalism allows us to precisely quantify the cost of implementing particular changes in the requirements, and to choose the most flexible implementation policy to implement these changes. Our formalism can be used at various levels of granularity (e.g., class level and method level). This implies that it can also be used during the design phase, when not all implementation details are known yet.

In the future we even plan to apply our ideas at an architectural level (as opposed to a design level), by considering changes to *architectural styles* [14] instead of design patterns. This will allow us to adapt existing architectural cost estimation models [9] to an evolution context.

7 Acknowledgements

This research was carried out in the context of the scientific network RELEASE financed by the ESF. We thank Jeff Reynolds for his useful comments and for his contribution to the definition of evolution complexity. We also thank Bart Dubois for proofreading this paper.

References

[1] Chapin, N., J. Hale, K. Khan, J. Ramil and W.-G. Than, *Types of software evolution and software maintenance*, Journal of software maintenance and evolution **13** (2001), pp. 3–30.

[2] Chaumun, M. A., H. Kabaili, R. K. Keller and F. Lustman, *A change impact model for changeability assessment in object-oriented software systems*, Science of Computer Programming **45** (2002), pp. 155–174.

[3] Chidamber, S. R. and C. F. Kemerer, *A metrics suite for object oriented design*, Transactions on Software Engineering **20** (1994).

[4] Fenton, N. E. and S. L. Pfleeger, "Software Metrics: A Rigorous and Practical Approach," PWS Publishing Company, 1997.

[5] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, 1995.

[6] IEEE 610.12-1990, "Standard Glossary of Software Engineering Terminology," IEEE Standards Software Engineering: Customer and Terminology Standards **1**, IEEE Press, 1999 .

[7] ISO 9126, "Information technology - software product evaluation - quality characteristics and guidelines for their use." ISO/IEC, 1991 .

[8] Jørgensen, M., *Experience with the accuracy of software maintenance task effort prediction models*, IEEE Trans. Software Engineering **21** (1995), pp. 674–681.

[9] Kazman, R., J. Asundi and M. Klein, *Quantifying the costs and benefits of architectural decisions*, in: *Proc. Int'l Conf. Software Engineering*, 2001, pp. 297–306.

[10] Lehman, M. M., J. F. Ramil, P. Wernick, D. E. Perry and W. M. Turski, *Metrics and laws of software evolution - the nineties view*, in: *Proc. Int'l Symp. Software Metrics* (1997), pp. 20–32.

[11] Li, L. and A. Offutt, *Algorithmic analysis of the impact of changes to object-oriented software*, in: *Proc. Int'l Conf. Software Maintenance* (1996), pp. 171–184.

[12] McCabe, T., *A software complexity measure*, Transactions on Software Engineering **2** (1976), pp. 308–320.

[13] Ramil, J. F. and M. M. Lehman, *Metrics of software evolution as effort predictors - a case study*, in: *Proc. Int'l Conf. Software Maintenance*, 2000, pp. 163–172.

[14] Shaw, M. and D. Garlan, "Software Architecture — Perspectives on an Emerging Discipline," Prentice Hall, 1996.

[15] Sneed, H., *Estimating the costs of software maintenance tasks*, in: *Proc. Int'l Conf. Software Maintenance* (1995), pp. 168–181.

[16] Urquhart, A., *Complexity*, in: L. Floridi, editor, *The Blackwell Guide to Philosophy of Computing and Information* (2004).

[17] Zuse, H., "Software Complexity: Measures and Methods," De Gruyter, 1991.

Evolution Through Architectural Reconciliation

Paris Avgeriou [1] Nicolas Guelfi [2] Gilles Perrouin [3]

Software Engineering Competence Center
Faculty of Science, Technology and Communication
University of Luxembourg
6, rue Richard Coudenhove-Kalergi
Luxembourg-Kirchberg, Luxembourg

Abstract

One of the possible scenarios in a system evolution cycle, is to translate an emergent set of new requirements into software architecture design and subsequently to update the system implementation. In this paper, we argue that this form of forward engineering, even though addresses the new system requirements, tends to overlook the implementation constraints. An architect must also reverse-engineer the system, in order to make these constraints explicit. Thus, we propose an approach where we reconcile two architectural models, one that is forward-engineered from the requirements and another that is reverse-engineered from the implementation. The final reconciled model is optimally adapted to the emergent set of requirements and to the actual system implementation. The contribution of this paper is twofold: the application of architectural reconciliation in the context of software evolution and the formalization of both the specification and transformation of the architectural models. The architectural modeling is based upon the UML 2.0 standard, while the formalization of the architectural models and their transformation are based on set theory and first-order logic.

Key words: software architecture, architectural design, model transformation, architectural recovery, architectural reconciliation.

1 Introduction

It is nowadays well-established that evolution of software should not be taken lightly, as it may correspond up to 90% of the total lifecycle costs. As a re-

[1] Email: paris.avgeriou@uni.lu
[2] Email: nicolas.guelfi@uni.lu
[3] Email: gilles.perrouin@uni.lu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

sult, software evolution has emerged as a new and promising field of software engineering, which tackles the problems of software change and software maintenance [17,39]. Even though research on software evolution, has been taking place for more than thirty years, it is only recently that concrete results have started to appear and the research community began to grasp the significance of the field [16]. Over the past years, some of the most important advances include: the eight laws of evolution [15], a phenomenology of software evolution [5,13,17], theories and practices on formalizing software evolution [16,23], tools that analyze and depict evolution of systems [39,30,34].

Research in software evolution strives to answer either the 'how' or the 'what' and 'why' [16,17,28]. The 'how' concerns the methods, practices and tools for evolving a system, in particular for synchronizing three distinct entities: 1) a model of the real world that is constantly changing, e.g. a domain model for an e-business system, 2) a specification of the system, e.g. the system's software architecture, and 3) the system implementation. The 'what' and 'why' observe the phenomenon of software evolution, trying to identify its causes and its internal workings.

This paper deals with the 'how' of software evolution, and in specific with keeping the system implementation in synchronization with the real-world model through a system specification. The real-world model might potentially range from a full domain model or business model of the entire business system to a minimal requirements specification for the software system under development. However, for the purposes of this paper we are only interested in the software requirements specification and consider that requirements are indeed well-defined. As far as the system specification is concerned, it is a key factor in evolution, since it formalizes the description of the system to be built and bridges the abstract real-world domain model and the system implementation [16,28]. In our approach we adopt software architecture as a form of system specification, since it has been proposed as an ideal abstraction to support system evolution [3,11,12,33].

A typical scenario of architecture-based evolution is to design the system's architecture in order to address the new set of requirements, and then realize this architecture into the system implementation [3]. The problem with this forward engineering approach is that the architectural design takes into account only the set of requirements and not the existing system implementation. As a consequence, implementation constraints are overlooked in the architecture, which in turn cannot be properly implemented into code. Apparently the system implementation contains implicitly a large numbers of design decisions that are 'hidden' in the code and can potentially contradict the new set of design decisions that emerge in the architecture. Even if we do look at the system and try to identify implementation constraints, we will probably fail to identify everything unless we reverse-engineer the system. This paper proposes an approach to tackle this problem by applying architectural reconciliation. In particular, the architectural model that is used to develop the

next system release, is derived from reconciling two architectural models, one that is forward engineered from the requirements specification and a second that is reverse-engineered from the system implementation. In this sense, the reconciled architecture will not only address the new requirements, but it will also take under consideration the implementation constraints. The documentation of software architectures is based on the UML 2.0 standard and formal methods.

The rest of the paper is organized as follows: Section 2 provides the details of the proposed approach for evolution through architectural reconciliation, including the informal and the formal technique used for architectural description. Section 3 illustrates the application of the approach through a case study while Section 4 presents some related research work with respect to architectural reconciliation. Finally Section 5 wraps up with some conclusions.

2 Architectural Reconciliation

2.1 The Process of Reconciliation

At the beginning of an evolution cycle we have an existing system implementation as well as a new set of requirements. The process of their reconciliation is comprised of three steps and is graphically illustrated in Figure 1.

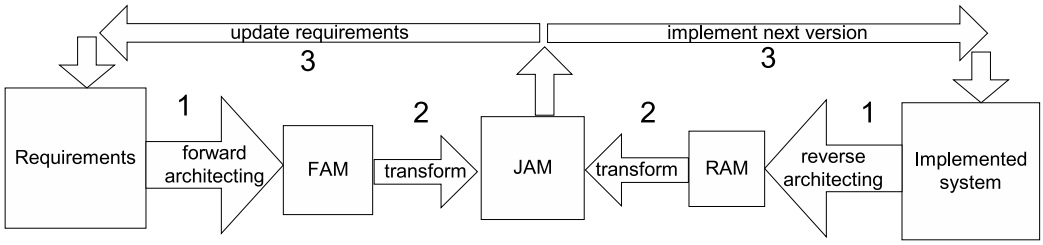


Fig. 1. Architectural Reconciliation for One Evolution Cycle

At the first phase we inspect the implementation code, and reverse-architect the implemented system in order to recover its architecture, which we name the **Reverse Architectural Model** or **RAM**. We do not prescribe a specific reverse-architecting approach, though there are a few such techniques and tools proposed, such as those in [10,24,30,34,38,39]. The RAM is mainly focused on identifying implementation constraints that may play a significant role during the system evolution. These constraints are usually expressed in the form of natural language and accompany the UML diagrams or the formal models. At the same phase, we use the new Requirements specification to design the *ideal* architecture of the system, which we name the **Forward Architectural Model** or **FAM**. This forward-engineering design of the architecture can be performed by following any architecture-driven software development process. It is important to stress that the FAM derives clearly from

the requirements side and strives to ideally satisfy the new requirements. The implementation constraints are not given much attention here, since they will be dealt with during the reconciliation in the next phase.

The second and most crucial phase is to bridge the RAM and the FAM into the **Joint Architectural Model** or **JAM**, which must satisfy both the new set of requirements and the implementation constraints. This is achieved by performing a transformation, that accepts the RAM and the FAM as inputs and produces the JAM as the output. In specific, the architect must go through the following steps:

(i) **Identify the implementation constraints that contradict the RAM.** These implementation constraints derive from the design decisions that the architect took during the previous evolution cycle and contradict the new design decisions taken in the FAM. The constraints may appear in any form, though we simply propose the use of natural language in combination with UML or formal models. This is because the software architecture community is still trying to tackle the problem of representing precisely local or global architectural constraints [21].

(ii) **Resolve the problems caused by the implementation constraints.** In order to resolve each such problem, the architect needs to decide between one, or a combination of the following actions:
   (a) Keep the part of the FAM and delete the part of the RAM that causes the problem.
   (b) Keep the part of the RAM and delete the part of the FAM that causes the problem.
   (c) Come up with a compromising solution that mixes both parts. In this case some of the elements from both models may be deleted, others may be retained, while more elements may be added.

(iii) **Complete the JAM.** The resolution of the problems will probably have consequences to other architectural elements that were not themselves part of the problem. Therefore, the architect needs to take some last decisions with respect to keeping, deleting or modifying FAM and RAM elements that were affected by the reconciliation actions.

The final phase in this process concerns with building the new version of the system according to the JAM, and also updating the requirements so as they reflect the reconciliation results. Figure 2 depicts the proposed approach over several evolution cycles, and also shows the rest of the details of the evolution process. It is of paramount importance to emphasize that there is always feedback from the JAM to the next evolution cycle. Firstly, as aforementioned, the JAM is not only used to develop the next implementation version of the system but it is also used to update the requirements according to the results of the reconciliation. Secondly, the JAM, as the only valid architectural model of the system, is used as the starting point for the FAM and RAM of the next iteration: FAM_i and RAM_i are designed starting from JAM_{i-1} and

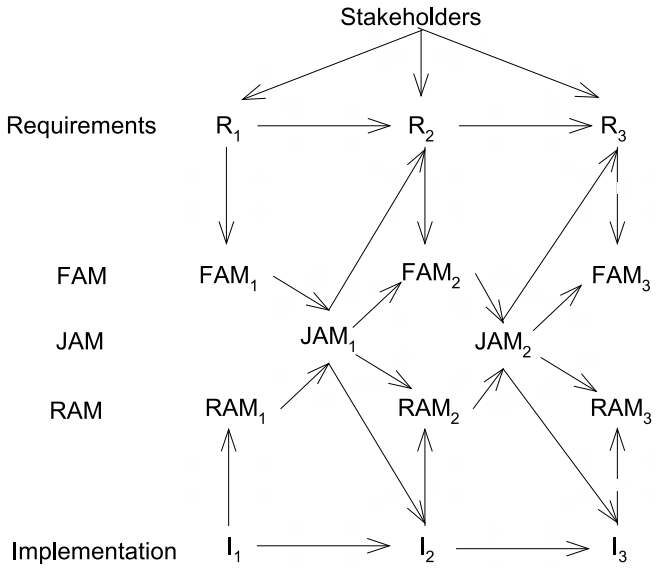


Fig. 2. Architectural Reconciliation over Several Evolution Cycles

by considering the new set of requirements R_i and the implementation I_i respectively.

The next two subsections propose two different ways for describing the architectural models: informally in UML 2.0, and formally using set theory and first-order logic.

2.2 Description of the Architectural Models in UML

An architectural description is comprised of multiple views [7,11,12,14]. In order to reduce the complexity of reconciliating two complex multiple-view architectural models, we have focused on the view that is considered to contain the most significant architectural information: the *component-and-connector* view [7]. This view deals with the system run-time by showing the *components*, which are units of run-time computation or data-storage, and the *connectors* which are the interaction mechanisms between components. However the same theory of reconciliation can equally well apply to the rest of the views.

The documentation of software architectures has been performed with the aid of **Architecture Description Languages (ADLs)**, which aim at formally representing software architectures [22]. Unfortunately these languages have never been broadly used in the industry and most of them lack support by appropriate tools. However the recent trend is the use of the widely accepted Unified Modeling Language as an ADL, which has become the 'lingua franca' of software design. We have adopted this approach and we have been working on the emergent UML 2.0 standard [27], which claims to provide large

support for modeling software architectures.
Our approach suggests therefore to model the component and connector view using UML 2.0 elements and especially those from the Composite Structures and Components packages. In specific the elements used for the component and connector view are:

(i) **Components**, which are specializations of classes and therefore have attributes and operations, but are also associated with provided and required interfaces. Components are also allowed to have an internal structure comprised of **properties** that in turn describe sets of instances of particular classes. Finally components may own ports that formalize their interactions points,

(ii) **Connectors**, which are either *assembly connectors* that connect the required interface of one component to the provided interface of a second, or *delegation connectors* that link the ports of a component to its internal parts,

(iii) **Interfaces**, which serve as contracts that components must comply with. An interface is either *provided* that describes a set of functionalities offered by a component, or *required* that describes a set of functionalities that a component expects from its environment.

(iv) **Ports**, which specify a distinct interaction point between the component that owns it and its environment or between the (behavior of the) component and its internal parts. Ports may specify required and provided interfaces of the component that owns them. A behavior port is a special case of a port, which sends all the incoming requests to the classifier that owns the port, rather than to its internal parts.

(v) **Classes**, that represent the constituents which realize the internal structure of components. These are not used in general-purpose class diagrams, but in composite structure diagrams, showing how the required and provided interfaces of a component delegate to or from its internal parts via the corresponding ports. Usually the composite structure diagrams do not contain the classes themselves, but sets of their instances in the form of properties.

Figure 3 illustrates the metamodel for the component-and-connector view, which is a subset of the UML 2.0 metamodel, and specifically the Components and Composite Structures packages. For reasons of simplicity some elements and other details have been omitted.

2.3 Formal Description of the Architectural Models

The syntax and semantics of UML are only semi-formally defined through a four layer meta-modeling architecture [26,27], that uses UML, natural language and OCL. The trend to formally specify the semantics of the language stems from the need to provide rigorous analysis and automatic transforma-

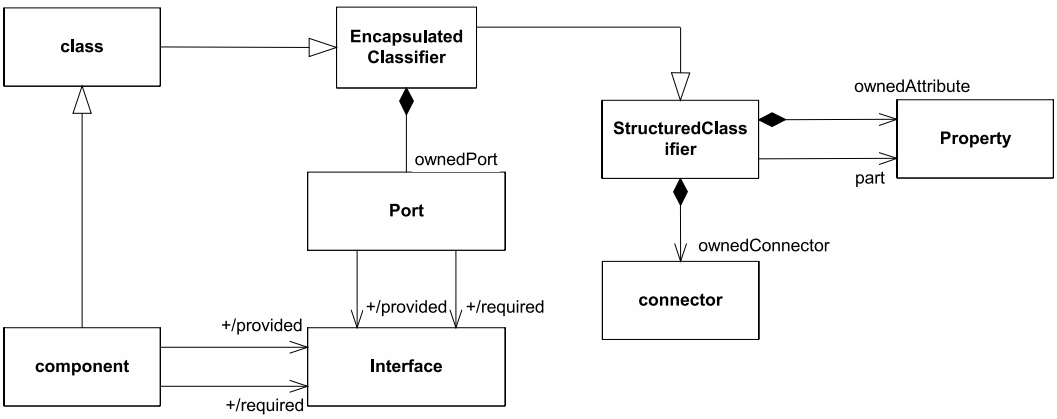


Fig. 3. UML 2.0 Metamodel for the Component and Connector View Elements (adapted from [27])

tions of UML models [18,35]. Different aspects of UML (e.g. class/component diagrams for modeling system structure, activity/statechart diagrams for modeling system behavior) require different forms of formalization. Process algebras [25] and Petri-Nets [2] allow for the formalization of system behavior while logic, sets and algebraic specifications [6,4,9] are used to formalize structural models. Graph grammars techniques [32] are also being used to formalize UML class and component diagrams [36,8].

We have concentrated on formalizing the component and connector view, that was described in the previous section, and makes use of UML 2.0 structural models. In specific we have defined a formal metamodel using set theory and first order logic, that specifies the semantics of models in the component and connector view. This formal metamodel is based upon a similar work by Richters [31], that provides a non-ambiguous definition of an object model (classes, attributes, operations, associations and a generalization hierarchy) and defines the semantics of OCL. We have extended that metamodel in order to define the elements of the component-and-connector view, namely components, ports, connectors, interfaces and properties, and specify their semantics. We therefore use this formal metamodel to formalize the reconciliation Re as a relation between three UML models satisfying a set of logical formulas φ that are inferred from the architect's tradeoff decisions: $Re = \{m, m = (FAM, RAM, JAM) | m \models \varphi\}$. The entire formal metamodel can't be included due to space restrictions, but a sample of these formulas are given both in logic and OCL in the next section.

3 A Case Study

The system that was chosen for this case study, is a popular open-source Learning Management System, named Ganesha [http://www.anemalab.org/ganesha], that supports e-learning in higher education and training institutes.

This system was chosen for two reasons: a) being an open-source project, its code can be inspected at will without the copyright issues of commercial systems; b) its simple PHP-based and medium-sized code makes it manageable and suitable for this kind of experiment.

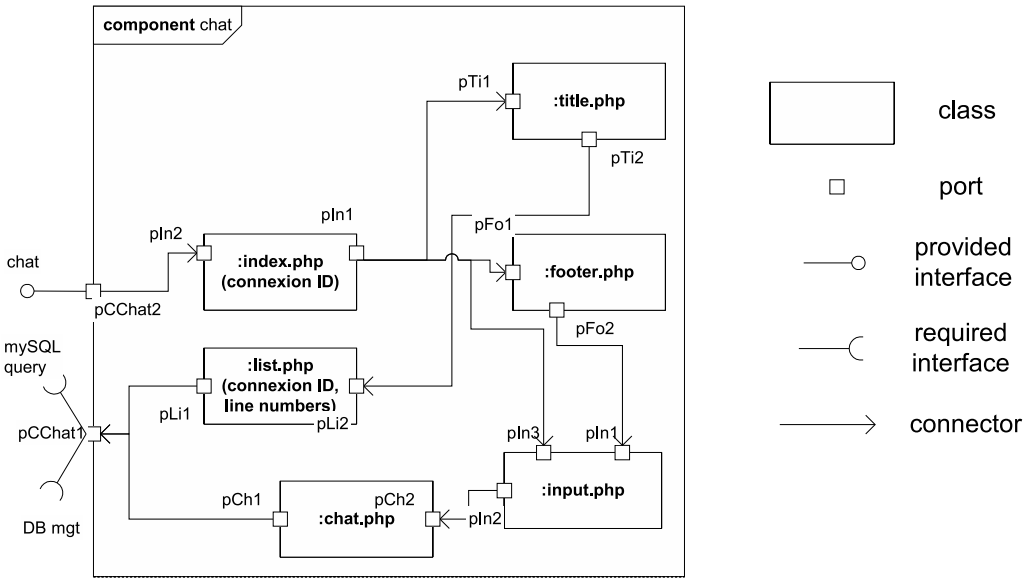


Fig. 4. The Reverse Architectural Model of the Chat Component

We have designed both the FAM and the RAM, as analytically described in [1]. For illustrative purposes, this section focuses on the reconciliation of a particular component of this system, the chat component, which allows for basic chat functionality, such as sending and receiving messages, and viewing the list of connected users. The reverse-engineered architectural model of this component was designed, based on the JAM of the previous iteration and the existing system implementation. As shown on Figure 4, the chat component is implemented through a number of PHP files, communicating through connectors, which are merely calls between them. These components "hide" design decisions that may serve as implementation constraints during evolution. In particular:

(i) **index.php** creates the overall frameset consisting of frames for displaying the messages, information about connected users and GUI elements for entering and formatting messages.

(ii) **title.php** checks which browser is used and calls list.php with the correct browser details.

(iii) **footer.php** is the frame at the bottom of the frameset which displays all the GUI elements for changing font colors and style, and notifies in-

put.php when such changes take place.

(iv) **list.php** updates the database for a new connected user and displays who has arrived and the set of currently connected members.

(v) **input.php** displays the GUI elements for entering messages, transforms these messages into the correct format according to footer.php and calls chat.php to display the messages.

(vi) **chat.php** is called by input.php if a message has been written in the text box, and displays that message along with information on who posted it. It also stores the messages to the database.

In the current evolution cycle, new requirements mandated the implementation of multiple chat rooms, that correspond to different topics of conversation, as well as the ability to exchange files between users. The architect therefore designed the FAM that satisfies these new requirements, starting from the JAM of the previous iteration. The forward architectural model, which is depicted on Figure 5, is naturally quite similar to the RAM, except for two new added components:

(i) **file.php** that implements the functionality for file management through an appropriate Graphical User Interface.

(ii) **room.php** that implements the functionality for changing rooms and displaying users' location. It is called by index.php and it also accesses the database to retrieve and store information about the room.

In order to perform the reconciliation we need to look at the two models, and try to resolve the potential problems, caused by implementation constraints. In this particular case, there were three implementation constraints that caused problems with the FAM and the architect needed to tradeoff on how to best resolve them. We describe the architect's decisions in three forms: natural language, logic formulas and OCL. For the OCL part we consider a model composed of the package "evol" with three sub-packages containing model elements from FAM, RAM and JAM. The three implementation constraints and the corresponding reconciliation actions, which resulted to the joint model (Figure 6), are the following:

(i) The GUI for entering text already exists and is included in input.php. Ideally we would want to put the GUI for changing rooms in the same place. Therefore the code for GUI in room.php should be moved to input.php, which should forward this information to the former for implementing the application logic. So these two components are modified into input2.php and room2.php, while they are also connected through a connector for the information exchange.

Logic: $room.php \in Comps_{FAM}, input.php \in Comps_{RAM} \Rightarrow room2.php \wedge input2.php \in Comps_{JAM} \wedge \exists c \in Conns_{JAM}$ *with* $connends(c) = \{input2.php, room2.php\}$

OCL:

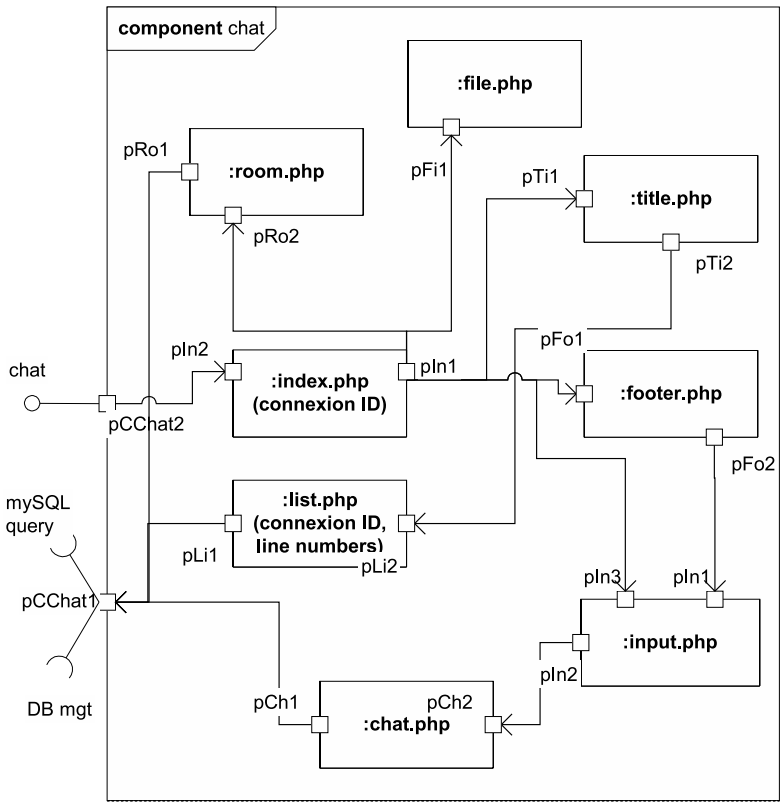


Fig. 5. The Forward Architectural Model of the Chat Component

```
context evol inv:
let pjam : Package = self.containedElements->select(Package)->
select(p|p.name='JAM')
in
self.containedElements->select(Package)->select(p|p.name='FAM')
.allContainedElements(Component)->exists(p|p.name='room.php')
and
self.containedElements->select(Package)->select(p|p.name='RAM')
.allContainedElements(Component)->exists(p| p.name='input.php')
implies
pjam.allContainedElements(Component)->exists(p|p.name='room2.php'
and p.Feature.Port->includes(pRo3))
and
pjam.allContainedElements(Component)->exists(p|p.name='input2.php'
and p.Feature.Port->includes(pIn2))
and
pjam.allContainedElements(Connector)->exists(c|c->ConnectorEnd.
role->size()=2 and c->connectorEnd.role->includesAll({pIn2,pRo3})
```

(ii) List.php displays the currently connected users in the right frame and that is where we also want to display information about what room the users are into. Thus room2.php needs to pass this information to list.php through a connector.
logic: $\exists c \in Conns_{JAM} \wedge connends(c) = \{room2.php, list.php\}$
OCL:

```
context evol inv:
self.containedElements->select(Package)->select(p|p.name='JAM')
.allContainedElements(Connector)->exists(c|c->ConnectorEnd.
role->size()=2 and c->connectorEnd.role->includesAll({pLi3,pRo4})
```

(iii) There is already a component in Ganesha that provides file management such as uploading and downloading files. This is not visible in the RAM for the chat component but in the RAM for the entire system. In this sense, file.php should reuse this component through its file management interface rather than implement this functionality from scratch. Therefore this component is slightly modified into file2.php and requires interface *file mgt* to operate, which is added as a required interface of the chat component.
logic: $file.php \in Comps_{FAM} \Rightarrow file2.php \in Comps_{JAM} \wedge \exists c \in Conns_{JAM}$, $connends(c) = \{file2.php, chat\} \wedge \exists file\ mgt \in Intfs_{JAM} \wedge file\ mgt \in requires(chat)$
OCL:

```
context evol inv:
let pjam : Package = self.containedElements->select(Package)->
select(p|p.name='JAM')
in
self.containedElements->select(Package)->select(p|p.name='FAM')
.allContainedElements(Component)->select(p|p.name='file.php')
implies
pjam.allContainedElements(Component)->exists(p|p.name='file2.php'
and p.Feature.Port->includes(\{pFi2\}))
and pjam.chat.Feature.Port->includes(\{pCChat3\})
and pjam.allContainedElements(Connector)->exists(c|(c->ConnectorEnd.
role->size()=2 and c->connectorEnd.role->includesAll({pFi2,pCChat3})
and
pjam.chat.required->includes(file mgt)
```

4 Related Work

The approach described in this paper has been based on research work with respect to bridging the gap between the system implementation and its requirements.
Perry and Wolf in [29] first introduced the architectural problems of erosion and drift, which express the phenomenon of having the implementation ar-

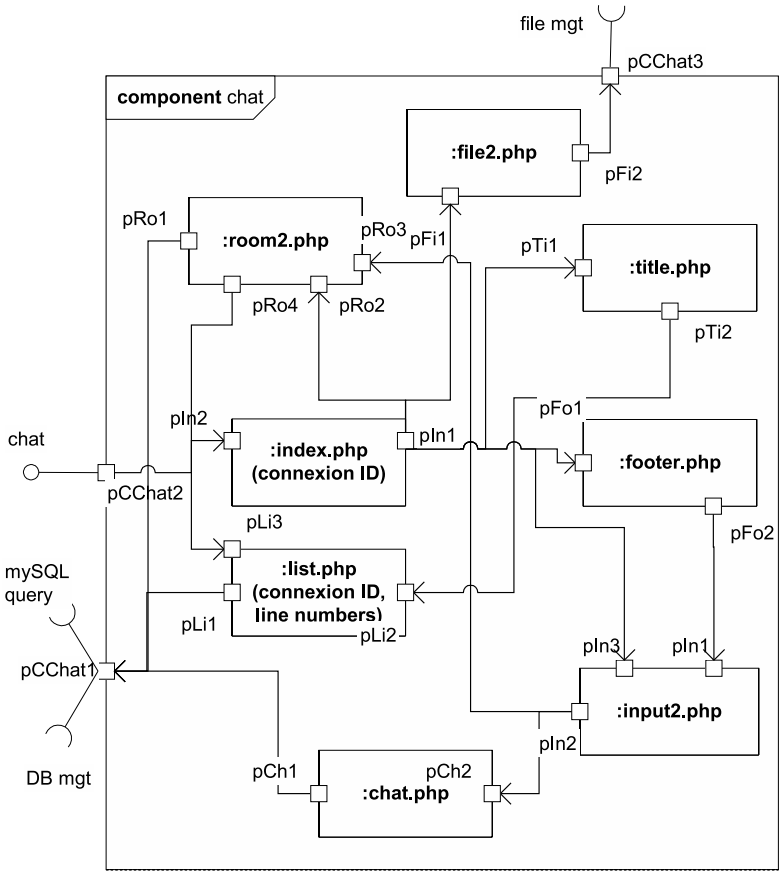


Fig. 6. The Joint Architectural Model of the Chat Component

chitecture driven away from the ideal architecture, either on purpose or due to indifference. In [37], Tran et al. introduced an architecture 'repair' technique for fixing this gap, by discovering and further eliminating the differences between the ideal architecture and the implementation architecture. They distinguish between forward repair where the implementation architecture is altered to match the conceptual, and reverse repair where the opposite takes place. Even though they have been mostly working with Open Source Software, where architectural drift is more likely to happen, they claim that their results can be generalized in commercial systems as well. They do not propose an approach for performing the design of the conceptual architecture but they do suggest tools such as those in [30,34] for reverse-architecting.

Roughly, the same problem has been dealt with in [19], where Medvidovic et al. propose the introduction of two intermediate steps: a) designing the 'discovered' architecture from the requirements and b) designing the 'recovered' architecture from the implementation. These two architectural models are then much easier bridged into the actual Architecture of the system. The 'discovery' of the architecture is performed using the CBSP method [20] that transforms the requirements into a handful of simple architectural elements

that are something between requirements and architecture. The 'recovery' of the architecture is performed using a blend of techniques that reverse-engineer the code and package the derived classes into architectural elements. The final bridging is performed manually by applying architectural styles to one of the two models and then mapping the second model to the outcome, or by first integrating the two models and then applying architectural styles.

Our own approach has been influenced by both of the aforementioned approaches since we have adopted the intermediate steps of FAM and RAM that [19] proposes and we have devised actions for the architectural reconciliation that are similar to those for *forward* and *reverse repair* [37]. Our work extends these approaches in the sense that we provide formalisms for the definition of the architectural models and subsequently their transformations in order to derive the final model. This formal transformation-based process is only part of our philosophy which states that everything can be considered as transformations between different artifacts, as will be explained in the final section.

5 Conclusions

In this paper we have argued that the evolution of a system cannot be performed effectively in a forward engineering style, e.g. transforming new system requirements into architecture and then into code. The problem is that the existing system implementation may place significant constraints to the new requirements and therefore must be taken under account. These implicit implementation constraints cannot be made explicit unless they are reverse-engineered. We have thus proposed to design two architectural models, the first based on the requirements and the second based on the existing implementation, and then reconciling these two models through logic-based transformations. The added value of our approach concerns two issues: the adoption of architectural reconciliation in the context of software evolution in order to overcome the problems of forward engineering and the formalization of both the architectural models and the transformations required to perform the reconciliation.

This approach constitutes a representative part of our holistic view on model-based software engineering. In particular we believe that transformation is a key mechanism during the entire development lifecycle and everything can be placed in the context of transforming artifacts into other artifacts. These transformations can take place: a) at a different level of abstraction, e.g. from design to code: b) at the same level of abstraction, e.g. from an architectural model to another architectural model that refines the former. The transformations are of paramount importance since they provide an association between the artifacts created; they are the conceptual 'glue' that binds everything together in a coherent set. This binding mechanism can provide the rationale for the decisions taken during the development process by tracing forward or backwards to the various artifacts.

We are currently elaborating other kinds of such transformations of the various artifacts, and we intend to end up with a fairly complete set of transformations that cover the entire process. We also intend to work on providing tool support for specifying and subsequently analyzing the formal models.

References

[1] Avgeriou, P., N. Guelfi and G. Perrouin, *Designing the forward and the reverse architecture of an e-learning system*, Technical Report TR-CST-04-04, Software Engineering Competence Center, University of Luxembourg (2004).

[2] Baresi, L., *Some preliminary hints on formalizing uml with object petri nets*, in: *Integrated Design and Process Technology*, 2002.

[3] Bosch, J., "Design and use of software architectures: adopting and evolving a product-line approach," ACM Press/Addison-Wesley Publishing Co., 2000.

[4] Breu, R., U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe and V. Thurner, *Towards a formalization of the unified modeling language*, Lecture Notes in Computer Science **1241** (1997).

[5] Buckley, J., T. Mens, M. Zenger, A. Rashid and G. Kniesel, *Towards a Taxonomy of Software Change*, Journal on Software Mainteance and Evolution: Research and Practice (2004).

[6] Clark, T. and A. Evans, *Foundations of the unified modeling language*, in: anonymous, editor, *2nd Northern Formal Methods Workshop* (1998).

[7] Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord and J. Stafford, "Documenting Software Architectures: Views and Beyond," Addison-Wesley, 2002.

[8] Engels, G. and R. Heckel, *Graph transformation as unifying formal framework for system modeling and model evolution*, in: *European Conference on Software Maintenance and Reengineering (CSMR 2001), International Special Session on Formal Foundations of Software Evolution*, Lisbon, Portugal, 2001.

[9] Favre, L. M., *A formal mapping between UML static models and algebraic specifications*, in: A. Evans, R. France, A. Moreira and B. Rumpe, editors, *Practical UML-Based Rigorous Development Methods - Workshop of the pUML-Group, UML 2001, Toronto, Canada*, LNI **P-7** (2001), pp. 113–127.

[10] Guo, G. Y., J. M. Atlee and R. Kazman, *A software archtiecture reconstruction method*, in: *WICSA-1*, San Antonio, TX, USA, 2001.

[11] Hofmeister, C., R. Nord and D. Soni, "Applied software architecture," Addison-Wesley Longman Publishing Co., Inc., 2000.

[12] IEEE, *Recommended Practice for Architectural Description of Software Intensive Systems*, Technical Report IEEE-std-1471-2000, IEEE (2000).

[13] Kemerer, C. F. and S. Slaughter, *An empirical approach to studying software evolution*, IEEE Trans. Softw. Eng. **25** (1999), pp. 493–509, IEEE Press.

[14] Kruchten, P., *The 4+1 view model of architecture*, IEEE Softw. **12** (1995), pp. 42–50.

[15] Lehman, M. M., *Laws of software evolution revisited*, in: *Proceedings of the 5th European Workshop on Software Process Technology* (1996), pp. 108–124.

[16] Lehman, M. M. and J. F. Ramil, *An Approach to the Theory of Software Evolution*, in: *International Workshop on Principles of Software Evolution IWPSE 2001*, Vienna, Austria, 2001.

[17] Lehman, M. M. and J. F. Ramil, *Software Evolution: Background, Theory, Practice*, Information Processing Letters **88** (2003), pp. 33–44.

[18] McUmber, W. E. and B. H. C. Cheng, *A general framework for formalizing uml with formal languages*, in: *Proceedings of the 23rd international conference on Software engineering* (2001), pp. 433–442.

[19] Medvidovic, N., A. Egyed and P. Gruenbacher, *Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery*, in: *STRAW'03 : Second International SofTware Requirements to Architectures Workshop at ICSE'03*, Portland, OR, USA, 2003, pp. 61–68.

[20] Medvidovic, N., A. Egyed and P. Gruenbacher, *Reconciling software requirements and architectures with intermediate models*, Journal for Software and Systems Modeling (SoSyM) (2004).

[21] Medvidovic, N., D. S. Rosenblum, D. F. Redmiles and J. E. Robbins, *Modeling software architectures in the unified modeling language*, ACM Trans. Softw. Eng. Methodol. **11** (2002), pp. 2–57.

[22] Medvidovic, N. and R. N. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Trans. Softw. Eng. **26** (2000), pp. 70–93.

[23] Mens, T. and M. Wermelinger, *Formal foundations of software evolution: workshop report*, SIGSOFT Softw. Eng. Notes **26** (2001), pp. 27–29, ACM Press.

[24] Mikic-Rakic, M., N. R. Mehta and N. Medvidovic, *Architectural style requirements for self-healing systems*, in: *1st Intl. Workshop on Self-Healing Systems*, Charleston, 2002.

[25] Ng, M. Y. and M. Butler, *Towards formalizing UML state diagrams in CSP*, in: A. Cerone and P. Lindsay, editors, *1st IEEE International Conference on Software Engineering and Formal Methods*, 2003, pp. 138–147.

[26] OMG, *UML 2.0 infrastructure specification*, Technical Report ptc/03-09-15, Object Management Group (2003).

[27] OMG, *UML 2.0 superstructure final adopted specification*, Technical Report ptc/03-08-02, Object Management Group (2003).

[28] Perry, D. E., *Dimensions of software evolution*, in: H. A. Müller and M. Georges, editors, *Proceedings of the International Conference on Software Maintenance* (Victoria, B.C., Canada; September 19-23, 1994), 1994, pp. 296–303.

[29] Perry, D. E. and A. L. Wolf, *Foundations for the study of software architecture*, ACM SIGSOFT Software Engineering Notes **17** (1992).

[30] Portable Bookshelf Website, http://www.swag.uwaterloo.ca/pbs/.

[31] Richters, M., "A Precise Approach to Validating UML Models and OCL Constraints," Ph.D. thesis, Universität Bremen, Fachbereich Mathematik und Informatik, Logos Verlag, Berlin, BISS Monographs, No. 14 (2002).

[32] Rozenberg, G., "Handbook of graph grammars and computing by graph transformation: volume I. foundations," 98-102288-48, World Scientific Publishing Co., Inc., 1997.

[33] Shaw, M. and D. Garlan, "Software Architecture: Perpectives on an Emerging Discipline," Prentice Hall., 1996.

[34] SHriMP website, http://shrimp.cs.uvic.ca/.

[35] Smith, J., M. K. Kokar and K. Baclawsi, *Formal verification of UML diagrams: A first step towards code generation*, in: A. Evans, R. France, A. Moreira and B. Rumpe, editors, *Practical UML-Based Rigorous Development Methods - Workshop of the pUML-Group, UML 2001, Toronto, Canada*, LNI **P-7** (2001), pp. 224–240.

[36] Tenzer, J., *A formal semantics of UML class diagrams based on transformation systems*, Technical Report 2001/09, Technische Universität Berlin (2001), publication of Master Thesis.

[37] Tran, J. B., M. W. Godfrey, E. H. S. Lee and R. C. Holt, *Architecture repair of open source software*, in: 8^{th} *IEEE International Workshop on Program Comprehension (IWPC 2000)* (2000).

[38] Tran, J. B. and R. C. Holt, *Forward and reverse architecture repair*, in: *CASCON'99*, Toronto, 1999, pp. 15–24.

[39] Tu, Q. and M. W. Godfrey, *An Integrated Approach for Studying Architectural Evolution*, in: *Intl. Workshop on Program Comprehension (IPWC-02)*, Paris, France, 2002.

Towards an Integrated View on Architecture and its Evolution

Martin Pinzger [1] Harald Gall [2]

Department of Informatics, University of Zurich

Michael Fischer [3]

Distributed Systems Group, Vienna University of Technology

Abstract

Information about the evolution of a software architecture can be found in the source basis of a project and in the release history data such as modification and problem reports. Existing approaches deal with these two data sources separately and do not exploit the integration of their analyses. In this paper, we present an architecture analysis approach that provides an integration of both kinds of evolution data. The analysis applies fact extraction and generates specific directed attributed graphs; nodes represent source code entities and edges represent relationships such as accesses, includes, inherits, invokes, and coupling between certain architectural elements. The integration of data is then performed on a meta-model level to enable the generation of architectural views using binary relational algebra. These integrated architectural views show intended and unintended couplings between architectural elements, hence pointing software engineers to locations in the system that may be critical for on-going and future maintenance activities. We demonstrate our analysis approach using a large open source software system.

Key words: software evolution analysis, software architecture, architectural views

1 Introduction

Higher-level views on the architecture of software systems aid engineers in evolving and maintaining software systems. Typically, these views are depicted as graphs whereas nodes represent the architectural elements and edges

[1] Email:pinzger@ifi.unizh.ch
[2] Email:gall@ifi.unizh.ch
[3] Email:M.Fischer@infosys.tuwien.ac.at

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

the relationships between them. In particular, relationships represent dependencies between architectural elements that lead to *coupling* between these elements. In theory, strongly coupled architectural elements are more likely to be modified together than are loosely coupled elements. Therefore, architectural designs concentrate on encapsulating common behavior within an architectural element, consequently increasing cohesion and lower coupling [2].

An *architectural element* in the context of our work is a *software module* that results from the decomposition of a software system into implementation units. According to Clements et al. [6] we refer to a software module as an implementation unit of software that provides a coherent unit of functionality. Modules present a code-based way of considering the system [2].

However, in practice evolution often draws a different picture revealing couplings between architectural elements that were not intended by the architectural design. Reasons for this are manifold: shortcomings in the initial design; the architecture has not been implemented in the way it was designed; or architecture drift due to frequent modifications to the implementation. In fact, these dependencies—for example constant changes crossing module boundaries—hinder the effective maintenance and evolution of software systems. Therefore, locating them in the current implementation to facilitate the application of directed refactorings to resolve these couplings would be beneficial.

In this paper we focus on analyzing dependencies between architectural elements and introduce the architecture analysis approach *ArchEvo*. ArchEvo enriches source code models extracted from source code and execution traces with logical coupling data obtained from configuration management systems [11,12]. We refer to logical coupling as: *Two source code entities (e.g. files) are logically coupled if a modification to the implementation affected both source code entities over a significant number of releases.*

The data sources are integrated into a common directed attributed graph from which ArchEvo abstracts higher-level views using architecture recovery [15]. In the analysis of the dependencies between architectural elements ArchEvo correlates both types of abstracted coupling relationships and shows strongly coupled elements as-implemented but also verifies these couplings by release history data. Consequently, the architectural views computed by ArchEvo provide an integrated view on the architecture and its evolution that points software architects and engineers to shortcomings in the design and implementation that should undergo directed refactorings.

The remainder of this paper is organized as follows: In Section 2 we introduce the architecture analysis approach ArchEvo and describe the building of the integrated fact repository and the abstraction of higher-level views. Section 3 describes our findings concerning the coupling relationships with a selected set of software modules and features of the open source web browser Mozilla. In Section 4 we present related work and Section 5 draws some conclusions and indicates future work.

2 ArchEvo Approach

Analyzing the dependencies between architectural elements is a key issue when analyzing the architecture of software systems. Recent research in analyzing these dependencies (i.e. coupling) concentrated on information obtained from source code and the running system. Briand et al. reported on the different measurements and described a framework of coupling measurements between classes and objects [3]. In our recent research we concentrated on investigating configuration management data including version, change, and defect data to obtain information about logical couplings (i.e. hidden dependencies) between source code units [10,12].

The ArchEvo approach presented in this paper is a combination of both approaches mentioned before and extends them by analyzing coupling relationships on the architectural level. Figure 1 depicts the process followed by ArchEvo. The process steps are described in the following subsections.

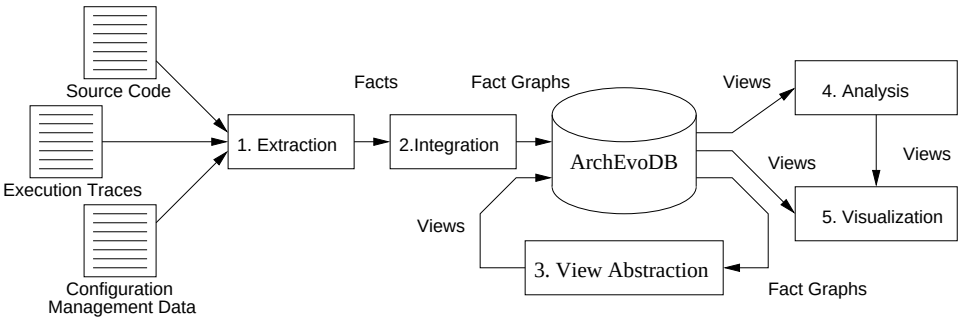


Fig. 1. ArchEvo architecture evolution analysis process.

2.1 Fact Extraction

Implementation specific data (i.e. facts) is obtained by applying static and dynamic analysis techniques including parsing and profiling. Parsing delivers static source code models that contain the source code specific entities such as files, packages, classes, methods, and attributes and the dependencies between them. Dependencies are file includes, class inherits and aggregates, method calls and overrides, and variable accesses. Profiling delivers run-time data (i.e. method call sequences) for an executed scenario and complements static source code models.

Release history data of a software project is obtained from configuration management systems in the form of modification reports. They are generated by versioning systems such as cvs [5] or Subversion [7] and deliver data about changes made to source files. These reports are parsed for relevant data used to identify logical coupling relationships. Following our definition of logical coupling we establish a logical coupling relationship between two entities if there is a modification report that references both entities [11]. Currently, logical coupling is detected on the level of source files which is sufficient for

our current approach. However, the integration of logical coupling on a more fine-grained level would be beneficial and is subject to future work.

As mentioned before different tools are used to obtain facts from a software system each using its own output format. For instance, static source code models and execution traces are stored in an ASCII file as directed attributed graphs (fact graphs in Figure 1). ArchEvo uses the Rigi Standard Format (RSF) for storing these graphs. Nodes represent extracted source code entities (e.g. files, classes, methods, attributes) and edges represent the relationships between them (e.g. includes, inherits, invokes, accesses).

Facts obtained from configuration management systems are stored to a relational database. To facilitate a common access of both data sources they have to be integrated into a common repository which is the *ArchEvoDB*. Because the ArchEvo abstraction approach needs directed attributed graphs the configuration management data stored in the relational database is converted to a fact graph. Nodes of this graph represent source files and modification reports and edges represent established couples relationships between source files. Next, these fact graphs are integrated into the ArchEvoDB that is a fact graph containing the source code, run-time and logical coupling data.

2.2 Data Integration

The two basic requirements for integrating the extracted heterogeneous fact graphs are: (a) facilities for extending the meta model to integrate new entity and relationship types; and (b) algorithms to map local to unique identifiers. Concerning the first requirement we use the FAMIX meta model for object-oriented programming languages [1] and extend it towards the inclusion of configuration management data, architectural views and metric data. Latter data is computed by the ArchEvo view abstraction algorithm described in the next subsection. The second requirement is fulfilled by our integration tool that maps locally unique identifiers (within a data file) to identifiers unique within the repository (ArchEvoDB).

Each fact graph is read by the integration tool that for each entity and relationship contained in the fact graph finds out about its identifier in the repository and if not exists computes a new one. Using the unique identifiers the new facts (nodes, edges, and attribute records) are added to the ArchEvoDB. The result is a common repository containing the integrated fact graph that forms the basis for the on-going abstraction and analysis tasks.

2.3 View Abstraction

In this step architectural views are abstracted from the integrated fact graph. ArchEvo supports abstraction to different levels of abstraction whereas the level is specified by the user. The abstraction algorithm used by ArchEvo is based on the approach presented by Holt et al. in [13], but extends it by computing measures for abstracted elements and relationships. An approach

similar to Holt's also has been described by Feijs et al. in [9].

Relationships between architectural elements and abstraction measures are computed using binary relational algebra. Currently, we use the grok tool [8] for calculating the binary relations because grok is able to handle extracted and integrated fact graphs in RSF format. However, the abstraction of attributes of relationships is not straight forward with grok, hence we implemented a workaround to handle this problem: For instance, the attribute values of lower-level relationships that form an abstracted relationship are summed up. Ongoing work is concerned with storing fact graphs in a relational database and use the standard query language SQL instead of grok.

Algorithm 1 defines the ArchEvo abstraction algorithm that is applied to the directed attributed fact graph.

Algorithm 1 *ArchEvo abstraction algorithm*

```
1:  foreach entity pair (A,B) do
2:      setA := entities contained by A
3:      setB := entities contained by B
4:      relsAB := relationships of type T between setA and setB
5:      if #relsAB >0 then
6:          rel := create relationship of type T between A and B
7:          measures := compute abstraction measures of rel
8:      end if
9:  end foreach
```

Having selected a relationship type to be abstracted the algorithm processes each pair of higher-level entities and first computes the two sets of entities (e.g. methods) contained in `A` and `B` (line 2,3). Next, the relationships of type `T` between the entities of set `A` and set `B` are queried (line 4) from the graph. If there is at least one relationship between any two lower-level entities of set `A` and `B` then an *abstracted* relationship between `A` and `B` is established (line 6). Measures concerning the number of affected lower-level relationships and entities are computed and stored in attributes of the new relationship (line 7). For instance, the number of modification reports making up a logical coupling is summed up and stored in the `refcount` attribute of the abstracted coupling relationship. Table 1 lists the measures computed for abstracted relationships (these measures also apply to other levels of abstraction).

Basically, ArchEvo distinguishes between direct and indirect dependencies whereas indirect stands for transitive. For both kinds of dependencies the number of involved source code entities are computed. Resulting measures reflect the weight of abstracted relationships and consequently quantify the coupling between architectural elements. They are used in the analysis of the dependencies between architectural elements.

Table 1. Measures computed for relationships abstracted to the module-level.

Measurement	Description
nrRelsDirect	# of abstracted direct lower-level relationships
nrRelsIndirect	# of abstracted indirect lower-level relationships
nrAdirectB	# of source code entities of direct relationships in module A
nrAindirectB	# of source code entities of indirect relationships in module A
nrBdirectByA	# of source code entities of direct relationships in module B
nrBindirectByA	# of source code entities of indirect relationships in module B
refcount	# of modification reports of a logical coupling relationship

2.4 Analysis

The goal of the analysis step is to indicate strongly coupled elements and to provide clues why these elements have such a strong coupling. The data used for this analysis is stored in the abstracted views. They contain the architectural elements (nodes), the coupling relationships (edges), and the coupling measures (attributes).

Coupling measures are stored in attributes of (abstracted) relationships. For instance, the number of method calls is stored in the `nrRelsDirect` attribute of an abstracted `invokes` relationship. For an abstracted `couples` relationship the number of modification reports is stored in the `refcount` attribute. Based on these attributes ArchEvo uses graph queries to determine the relationships of interest and the corresponding architectural elements.

Graph queries are implemented using a combination of binary relational algebra and Perl scripts. For example, to determine the elements with the strongest logical coupling ArchEvo applies a query to the `refcount` attribute of `couples` relationships that have a value greater than a given threshold.

For the correlation of source code coupling with logical coupling relationships ArchEvo ranks each relationship with respect to the computed average or maximum of a given relationship attribute (e.g. `refcount`). The ranking values are represented in matrices one per attribute. Using statistical methods on the matrices the correlation between the different relationships is computed providing users with quantitative measures about the dependencies.

The result of the quantitative analysis are refined architectural views that facilitate an assessment of the current architecture and its evolution, as well as the identification of design shortcomings. They also provide good starting points for a more detailed analysis of architectural dependencies, for instance, by selecting two modules that are strongly coupled.

The detailed analysis that qualifies and verifies quantitative measures is performed on a finer-grained level of abstraction such as the file-level. Considering the reduced set of files of the selected higher-level entities (i.e. modules) the logical coupling relationships are qualified with respect to the source code

coupling that caused it. Next the results of the qualification are reflected back to the higher-level views to enrich them with more details. They direct to locations of design shortcomings that should be resolved to smoothen evolution and maintenance.

In the next section we describe our analysis of the open source web browser Mozilla [18].

3 ArchEvo Views

The outcome of the ArchEvo architecture analysis process are views that can be used by the user to identify starting points for minor changes on lower level or major re-design phases. To demonstrate our ArchEvo approach we applied it to the open source web browser Mozilla version 1.3a (released December 2002). At this time the Mozilla application suite comprised more than 10.400 source files in C/C++ containing about 3.700.000 source text lines distributed over 2.500 directories and more than 90 software modules. Starting from Mozilla's design documentation we focused our analysis on a selected set of software modules as architectural elements that implement the internal representation (i.e. content) and the layout of web pages. Table 2 lists the selected software modules together with corresponding source code directories containing their implementation. The mapping between modules and source code directories has been taken from Mozilla's design documentation.

Table 2. Selected Mozilla modules and their source code directories

Module	Source Directories
MathML	layout/mathml
New Layout Engine	layout/base, layout/build, layout/html
XPToolkit	content/xul, layout/xul
Document Object Model (DOM)	content/base, content/events, content/html/content, content/html/document, dom
New HTML Style System	content/html/style, content/shared
XML	content/xml, expat, extensions/xmlextras
XSLT	content/xsl, extensions/transformiix

Subgoals in our analysis were: (a) abstraction from the low-level information to the level of software modules; and (b) correlating the abstracted implementation specific relationships with the modification specific ones. The objective was to obtain measurements (sizes, weights) of different coupling dependencies between the selected software modules including source code but also logical couplings as listed in Table 1

Based on these views and measurements we analyzed the as-implemented architecture of these modules with respect to their maintainability and evolvability. These two related quality attributes of software systems are influenced

by the coupling between software modules. Basically, the stronger the coupling is the more effort has to be spent for maintaining and evolving the system [3].

The following sections report on our findings about the selected modules listed in Table 2.

3.1 Module View

The module view reflects the as-implemented design together with the release history information. The elements of the representation are software modules, their source code and logical coupling relationships.

The resulting graphs—different types of relationships can be selected for the graph generation—gives a first quantitative feedback about inter-module coupling. Figure 2 depicts invocations—represented as red/solid arcs—between the selected modules which are represented as gray boxes. Width and height of the boxes indicate the size of software modules in terms of number of global functions and methods (width) and global variables and attributes (height) of a module. The distance between two modules is determined by the number of logical couplings (i.e., pairwise changes) between these modules and indicated as straight, cyan/solid line. Since all modules are coupled with each other through "administrative noise", weaker couplings are omitted. As threshold we used 10% of the maxium coupling (`DOM` – `New Layout Engine`) which in turn comprises more than 48.000 pairwise modifications. The "administrative noise" mentioned above typically involves several hundred files. Messages left in the description field of these administrative modifications are for instance "license foo" (with 7.961 referenced files), "printfs and console window info

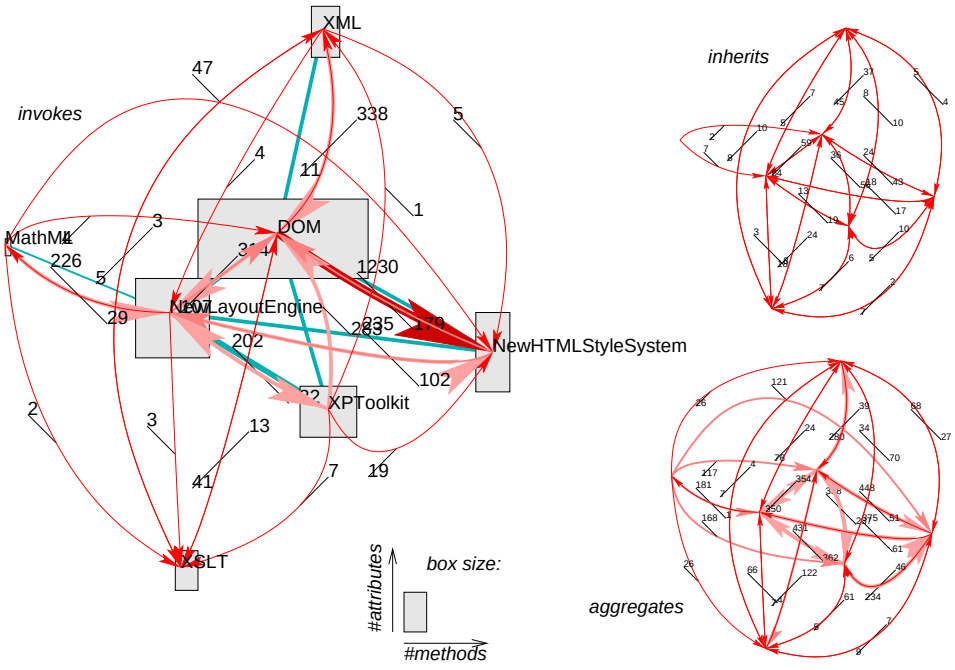


Fig. 2. Invokes, inherits and aggregates between selected modules

needs to be boiled away for release builds" (1.135 files), or "Clean up SDK includes" (888 files).

With this view one can easily spot the strong coupling between the three modules in the center of the graph (`New Layout Engine`, `DOM`, `XPToolkit`). Interesting to see is the high number of mutual calls between these modules. Consequently, when modifying one of these modules it is very likely that the other modules have to be touched.

The two small graphs in Figure 2 on the right side show the same coupling graph but the inherits and aggregates relationships. The strongest edges indicate a high correlation with the strong couplings between the modules. In both views the mutual dependencies can be observed as well.

As a result, abstracted module views pointed out locations of strong couplings that caused pairwise modifications of software modules. Directed refactorings can be applied to these locations to improve the design and reduce the pairwise modifications in the future. However, module views are abstract representations of underlying source code data. Therefore, deeper insights into the coupling dependencies are mandatory to refactor them.

3.2 A Detailed Module View

For a detailed evaluation of the coupling between software modules we selected the modules `New Layout Engine` and `XPToolkit`. The focus was on the source files of both modules that have the strongest coupling. These files represent the design critical source code entities. The resulting graph comprises six files and is depicted in Figure 3. It shows method invocations (red/solid arcs) gathered from the runtime data and the logical couplings between source files (straight cyan/solid lines).

The layout, i.e., the relative position of the boxes to each other, is defined by the number of logical couplings found between files. Actually, the highest coupling crossing the module boundaries exists between *nsPresShell.cpp* and *nsXULDocument.cpp* with 81 problem reports. The flags with the numbers of invocations attached to the arcs are always pointing from the caller and indicate the actual number of dynamic invokes found. For example, there are 4 calls from *nsXULElement.cpp* to *nsPresShell.cpp* and 3 calls in the other direction.

The central position of *nsCSSFrameConstructor.cpp* indicates a high degree of coupling with other files. This strong logical coupling is further strengthened by the method invoke relationships which cover all other files in this view. Therefore, this file is the most critical entity concerning evolving or maintaining the two modules.

Summarized, the case study showed the bottom-up abstraction of lower-level information to architectural views (i.e. module view). These views are mandatory to point out the modules that are most involved in pairwise changes. Next going top-down from architectural views to lower-level views

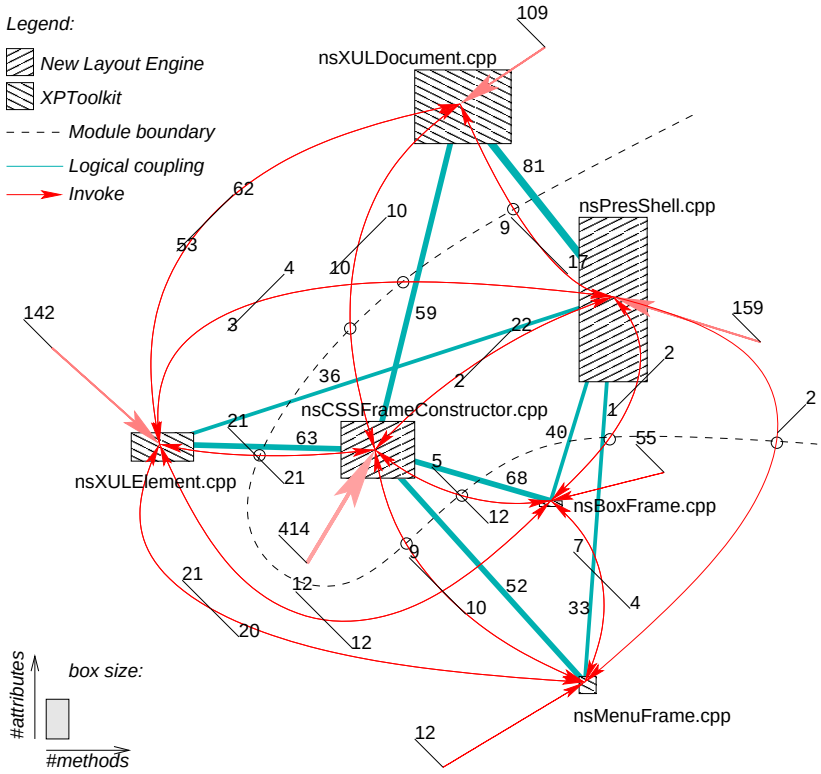


Fig. 3. Invocations between files (Modules `XPToolkit` and `New Layout Engine`)

the details making up and causing these logical couplings in the implementation are revealed. Knowing the critical source code entities the user can direct his refactoring activities to these entities to improve the *as-implemented design* of the system under study.

4 Related Work

Related work ranges from evolution analysis to architecture recovery and coupling analysis approaches. Concerning evolution analysis Zimmermann et al. inspected release history data of several software systems for logical coupling between source code entities [20]. They drew the conclusion that augmentation of architectural data with evolutionary information could reveal new otherwise hidden dependencies between source code entities. Even though a number of other work used release history data as well, a detailed evaluation of the correlation between source model entities and the properties of logical coupling is still missing.

Hsi and Potts [14] studied the evolution of user-level structures and operations of a large commercial text processing software package over three releases. Based on user interface observations they derived three primary views describing the user interface elements (morphological view), the operations a user can call (functional view), and the static relationships between objects

in the problem domain (object view). As this approach does not consider a thorough code analysis, user interface issues are usually not taken into account during code analysis, a fusion with methods regarding code and release history data would yield good results in feature evolution analysis.

In [12] we examined the structure of a *Telecommunications Switching Software* (TSS) over more than 20 releases to identify logical coupling between system and subsystems. As step towards combination of abstract concepts such as feature information and logical coupling we investigated in [10] the reflection of qualified release history data onto different source code model entities. Release history data comprised modification report plus problem tracking data. The source code model data representing different features were derived from source code using runtime information. The verification of properties of the underlying source code model such as aggregation etc. were beyond the scope of this work.

In [3] Briand et al. discussed a unified framework for coupling measurement in object-oriented systems based on source model entities. Based on this metrics they verified in [4] the coupling measurements on file level using statistical methods and logical coupling information based on "ripple effects" [19]. A classification of logical coupling information to verify the properties of coupling measurement has been omitted. In our approach we go further and use file level information to abstract onto higher architectural views such as module view as well.

Wilkie and Kitchenham investigated the correlation of coupling between objects (CBO) and change ripples of a C++ application [16,17]. Their work was focused on class level properties in contrast to our work which is primarily focused on higher, more abstract architectural views.

Concerning architecture recovery several related approaches exists such as, for example, described by Holt et al. in [13] and Feijs et al. in [9]. Both approaches deal with abstracting lower-level source code information. The abstraction algorithm used by ArchEvo is based on them. However, in extension to these approaches ArchEvo takes into account modification and problem report data. Further, ArchEvo concentrates on the computation of measures for abstracted elements and relationships.

5 Conclusions and Future Work

The ArchEvo approach combines information gained from static and dynamic analyses of the source code with release history data into specific views on different abstraction levels. The analysis applies fact extraction and generates specific directed attributed graphs; nodes represent source code entities and edges represent relationships such as accesses, includes, inherits, invokes, and coupling between certain architectural elements. The integration of data is then performed on a meta-model level to enable the generation of architectural views using binary relational algebra. These integrated architectural

views show intended and unintended couplings between architectural elements, hence pointing software architects to locations in the system that may be critical for on-going and future maintenance activities. Thus, ArchEvo's contribution is the abstraction of detailed source code model data and evolutionary information onto more abstract levels. This supports the reflection of the concrete implementation at its design level with focus on the coupling dependencies between architectural elements. Details of selected coupling dependencies are obtained by decreasing the level of abstraction. Consequently, the benefits of ArchEvo are in (1) graphically highlighting locations of design erosion in the as-implemented architecture that led to logical couplings; and (2) revealing the implementation details potentially causing them.

Further benefits of the ArchEvo approach are: (a) compact graphical representation of architectural data together with evolutionary information; (b) location of areas with frequent modifications; (c) providing good views onto dependencies between different elements of the source code model; (d) support for different views on arbitrary abstraction levels such as file-, component-, or module-level; and (e) quick identification of tightly coupled files or modules through their placement in the graphical representation. Finally, our approach has been validated using the large open source software project of Mozilla.

Interesting areas for future work are qualitative and quantitative analysis of the properties of logical and architectural coupling such as inter- and intra-module coupling, evaluation of properties of transitive dependencies, i.e., logical coupling but no direct architectural dependency between different source code model entities, extending this methodology to support change impact analysis of large scale software, integration of source code model deltas between different releases to automatically classify the type of modification such as interface changes, add/remove invocation relationship, aggregation etc. (statement level analysis). Another perspective is the integration of problem report data into the analysis process to deliver further hints for the search of error prone entities within the abstracted views.

Acknowledgments
We thank the Mozilla developers for providing all their data for this case study. The work described in this paper was supported by the Austrian Ministry for Infrastructure, Innovation and Technology (BMVIT), The Austrian Industrial Research Promotion Fund (FFF), and the European Commission in terms of the EUREKA 2023/ITEA projects CAFÉ and FAMILIES (http://www.infosys.tuwien.ac.at/Cafe/).

References

[1] "The FAMIX 2.0 Specification," 2.0 edition (1999), http://www.iam.unibe.ch/ scg/Archive/famoos/FAMIX/.

[2] Bass, L., P. Clements and R. Kazman, "Software Architecture in Practice," Addison-Wesley, 2003, 2nd edition.

[3] Briand, L. C., J. W. Daly and J. K. Wüst, *A unified framework for coupling measurement in object-oriented systems*, Journal of IEEE Transactions on Software Engineering **25** (1999), pp. 91–121.

[4] Briand, L. C., J. Wüst and H. Lounis, *Using coupling measurement for impact analysis in object-oriented systems*, in: *Proceedings of the IEEE International Conference on Software Maintenance* (1999), pp. 475–482.

[5] Cederqvist, P., "Version Management with CVS," (1992), http://www.cvshome.org/docs/manual.

[6] Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord and J. Stafford, editors, "Documenting Software Architectures: Views and Beyond," Addison-Wesley Professional, 2002.

[7] Collins-Sussman, B., B. W. Fitzpatrick and C. M. Pilato, "Version Control with Subversion," (2004), http://svnbook.red-bean.com/svnbook-1.1.

[8] Fahmy, H. and R. C. Holt, *Software architecture transformations*, in: *Proceedings of the IEEE International Conference on Software Maintenance* (2000), pp. 88–96.

[9] Feijs, L., R. Krikhaar and R. van Ommering, *A relational approach to support software architecture analysis*, Journal of Software Practice and Experience **28** (1998), pp. 371–400.

[10] Fischer, M. and H. Gall, *Visualizing feature evolution of large-scale software based on problem and modification report data*, Journal of Software Maintenance and Evolution **16** (2004).

[11] Fischer, M., M. Pinzger and H. Gall, *Populating a release history database from version control and bug tracking systems*, in: *Proceedings of the IEEE International Conference on Software Maintenance* (2003), pp. 23–32.

[12] Gall, H., K. Hajek and M. Jazayeri, *Detection of logical coupling based on product release history*, in: *Proceedings of the IEEE International Conference on Software Maintenance* (1998), pp. 190–198.

[13] Holt, R. C., *Structural manipulations of software architecture using tarski relational algebra*, in: *Proceedings of the IEEE Working Conference on Reverse Engineering* (1998), pp. 210–219.

[14] Hsi, I. and C. Potts, *Studying the evolution and enhancement of software features*, in: *Proceedings of the 2000 IEEE International Conference on Software Maintenance*, 2000, pp. 143–151.

[15] Pinzger, M., M. Fischer, M. Jazayeri and H. Gall, *Abstracting module views from source code*, in: *Proceedings of the IEEE International Conference on Software Maintenance* (2004).

[16] Wilkie, F. G. and B. A. Kitchenham, *Coupling measures and change ripples in c++ application software*, Journal of Systems and Software **52** (2000), pp. 157–164.

[17] Wilkie, F. G. and B. A. Kitchenham, *An investigation of coupling, reuse and maintenance in a commercial c++ application*, Journal of Information and Software Technology **43** (2001), pp. 801–812.

[18] *Mozilla open-source web browser*, http://www.mozilla.org.

[19] Yau, S. S., J. S. Collofello and T. MacGregor, *Ripple effect analysis of software maintenance*, in: *Proceedings of the IEEE International Computer Software and Applications Conference* (1978), pp. 60–65.

[20] Zimmermann, T., S. Diehl and A. Zeller, *How history justifies system architecture (or not)*, in: *Proceedings of the IEEE 6th International Workshop on Principles of Software Evolution* (2003), pp. 73–83.

Fresco: Flexible and Reliable Evolution System for Components

Yves Vandewoude [1,2] and Yolande Berbers [3]

Department of Computer Science
KULeuven
B-3000 Leuven, Belgium

Abstract

Fresco is a methodology that allows for the dynamic adaptation of component-oriented applications. Fresco aims to support the developer in the realization of dynamic adaptation throughout the entire life cycle of a component. At design time, a tool (DeepCompare) assists the programmer in the preparation of a component with live update functionality. At runtime, a middleware environment called Draco guides the replacement process itself and ensures that a component replacement is executed correctly.

In this position paper, the focus is on the design time support and the tool DeepCompare. After the functional development of a component, DeepCompare constructs a meta-model from both the old and the new component versions. These models are compared and equivalent data-structures are identified. This information is subsequently used to partially generate state transition functions. Possible benefits include the verification of the correctness of an update using component invariants and the estimation of the complexity of the upgrade in order to flag certain problem scenarios to the developer.

Key words: Software Evolution, Change Detection, Live Updates, State Transfer

1 Introduction

Research shows that over 80% of the cost of a software product is caused by maintenance, and more than 20% of the initial specifications of a product are considered outdated within a year after deployment ([15]). Keeping software up-to-date is a major problem that affects developers and users alike. The last

[1] Supported by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen)
[2] Email: yves.vandewoude@cs.kuleuven.ac.be
[3] Email: yolande.berbers@cs.kuleuven.ac.be

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

few years, the tendency arose to address this issue by modularising software with component-oriented methodologies. Applications are constructed by creating compositions of loosely coupled units of functionality: Components.

Reusing components shortens development time and increases robustness. For the user, however, the problem remains. Updating the software requires shutting it down and installing a new version, which can be a risky and expensive operation. A possible solution for this problem can be found in *live updates*: modifying the software at runtime. In practice however, these techniques are seldom used due to their high complexity and cost. A large portion of this complexity is caused by the state transfer between the old and the new version of the software. The transfer and conversion of the state of a component must be manually implemented. This is a tedious and error-prone task.

In this paper we present Fresco, which provides a practical approach for live updates. We begin with a general overview of the methodology in section 2. Since the focus of this position paper is on the design time aspect, the second part of this paper elaborates on the tool DeepCompare. The general architecture and functionality of the tool are presented in section 3. We illustrate the potential of our approach by processing a small example with DeepCompare. The example and its results are shown in section 4. Future work is discussed in section 5. References to important related work are given in section 6. We conclude this paper in section 7.

2 Adapting Component-based Applications

Fresco's main goal is to increase the applicability of live updates. It achieves this goal as follows:

Limited manual preparation: Fresco requires very little intervention of the programmer during the development of the component. A preprocessor is used to add an interface that can be used to access the internal structure of the component by a later version. This is accomplished by adding getter methods if these are not yet available and is fully automatic.

Tool support for component instrumentation: One of the most difficult issues during a component replacement is the transfer of state between the old and the new version of a component. The Fresco methodology places the responsibility of this transfer with the new component version. During the update, a reference to the old version is given to the newly created instance which interprets internal structures of its predecessor and subsequently imports its state. This functionality is far from trivial and the instrumentation of the new component is usually left to the programmer. Fresco is innovative in its ability to generate large portions of this transfer code (see sections 3 and 4).

Advanced Runtime Support: Fresco provides support for live updates at runtime. This aspect is implemented as an extension to the Draco middle-

ware environment. This *dynamic update module* (DUM) is responsible for the correct execution of the update at runtime. It puts the active component in an inactive [4] state and then loads and instantiates the new component version. The DUM subsequently passes on the old version to the new instance, which allows it to initialize itself with the state information from its predecessor (the methods for this functionality were added by DeepCompare in the instrumentation phase). Finally, the DUM reroutes connections from the old to the new component. Since the Draco middleware system and the dynamic update module are not the focus of this paper we refer to [19,20] for more information on the dynamic aspects of the Fresco methodology.

3 Design-time support

3.1 Direct vs. Indirect State Transfer

The Fresco evolution system is based on components that are designed according to the Cocones component methodology ([3,17]). These components are implemented as a group of tightly coupled Java-objects. The runtime system guarantees that a component is in an inactive state during the update (no methods are active). Therefore the state of the component is contained in the instance variables of the objects that make up its structure. In literature, two approaches exist to transfer state between versions:

Direct State Transfer: The implementation of the old version is used directly. It is the responsibility of the new version to interpret and convert the state from the previous version.

Indirect State Transfer: The old version exports its state in an abstract representation which is later used by the new version.

While the latter approach has the benefit that it allows easy upgrading when many different versions are involved, the technique strongly depends on the ability to construct an ontology that defines the abstract form. Therefore, this method is only used in specific and well defined fields such as protocol stacks (see [10]). Furthermore, the functionality to export its state in an abstract form must be implemented by each component, even if it is unsure whether it will ever be dynamically replaced.

Therefore, the first approach was used in the Fresco methodology. Since the instrumentation of the new component is complex and depends on the previous version, tool support is provided for this activity: DeepCompare.

3.2 DeepCompare

The generation of state transition functionality consists of two steps. First, equivalent data-structures must be identified. This information is then used

[4] More accurately: a quiescent state (see [11]).

to generate state transition functions.

Identification of Similarities
DeepCompare starts by parsing both versions of a component, and buiding a meta-model. It then compares all types of the two projects and constructs a similarity model. This model consists of a *change-description* for each couple of types. The similarity model is then used and updated by a chain of *harvesters*. Each harvester retrieves information from both component versions and updates the similarity model by adding *semantic links* to this model. As such, the similarity model represents all known similarities at each stage of the comparison proces. Each harvester encapsulates a different algorithm. Next to trivial matches (e.g. variables with identical name and type in the same class are considered equivalent), more complex matches can be derived. For instance, the current tool is capable of detecting added encapsulation, type renaming and variable movements between types (see section 4). Three types of algorithms are currently investigated:

(i) *Structural Algoritms* will identify new similarities given known similarities and the structure of both components. This principle is used in the context of Database scheme evolution by Lerner ([12]). An example of a harvester that uses a structural algorithm is a harvester that will search for class variables that have been moved up or down a class hierarchy.

(ii) Techniques from *Plagiarism Detectors* ([14,21]) can be used to identify similar structures between different programs. The resulting similarities can either be directly exploited (e.g. similar variable names or comments used in the same context), or be used indirectly by other harvesters (e.g. by identifying similar methods – information that can be exploited by structural algorithms).

(iii) *Refactoring Detectors* (e.g. [5]) exploit the principle that software evolves gradually. Due to the popularity of Extreme Programming ([2]), refactoring has gained much in importance in the field of reengineering. After a refactoring is found between different versions, underlying structure similarities can be identified. Detecting the encapsulation of variables in composite types is an example of refactoring detection.

While it is theoretically possible to implement DeepCompare using abstract syntax trees, we consider such representations too low-level. A meta-model is used to verify the correctness of the underlying code and to allow for easy inspection and manipulation of the underlying source code. As our components are implemented in Java, existing Java-based meta-models can be used. A number of meta-models exist in literature, each with a different philosophy and complexity. At the time of writing, JNome ([6]) is used as the underlying meta-model. However, since it is conceivable that other models may later appear to be more suitable, the actual meta-model used by the harvesters is abstracted using the adaptor pattern [7]).

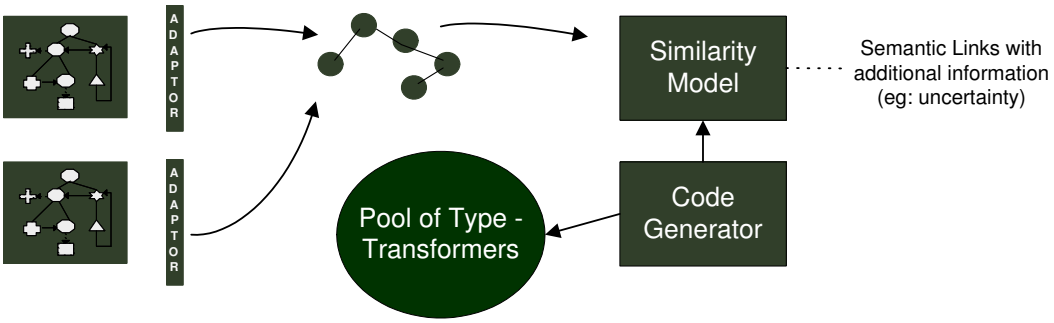


Fig. 1. Schematic Overview of DeepCompare

Generation of state transition code
In a second phase, DeepCompare will generate the transition function between two versions. The basic building block of this process is the *type transformer*. Each transformer converts a structure of a given type into another type (e.g. a transformer may convert an array into a Vector or vice versa). Type Transformers are implemented manually and added to a repository. DeepCompare will construct the state transition functionality by selecting the appropriate type transformers based on information contained in the similarity model. Composite types are transformed by recursively applying known transformers to the type. Newly added variables for which no equivalent is found in the previous version are by default initiated to the types default value. This behaviour can be altered by the user. The type transformation functionality of the tool is not yet fully implemented.

4 DeepCompare at work: a simple example

In general, it is impossible to correctly identify semantically equivalent structures between different versions of a component since not all semantic information is contained in source code. A typical example is the representation of a triangle. In version n, three points may be used while version $n+1$ uses two edges and an angle. Therefore, identification of similarities is in essence an interactive and semi-automatic process. At all times, the user of DeepCompare can guide the tool by adding or removing semantic links between types.

However, it is our belief that the majority of structures that make up the state of a component has sufficient similarity between versions that it can be detected by a tool. A small example is shown in figure 2. On this figure, rectangles represent the objects that make up the component. Ovals connected with these rectangles are instance variables associated with these objects. Blue and red items are part of the original/new version of the program respectively. Yellow types are part of the JDK and are considered to be fixed. Orange refers to primitive types.

The example consists of a `Line` with some associated methods. In the original, four instance variables are present representing the coordinates of

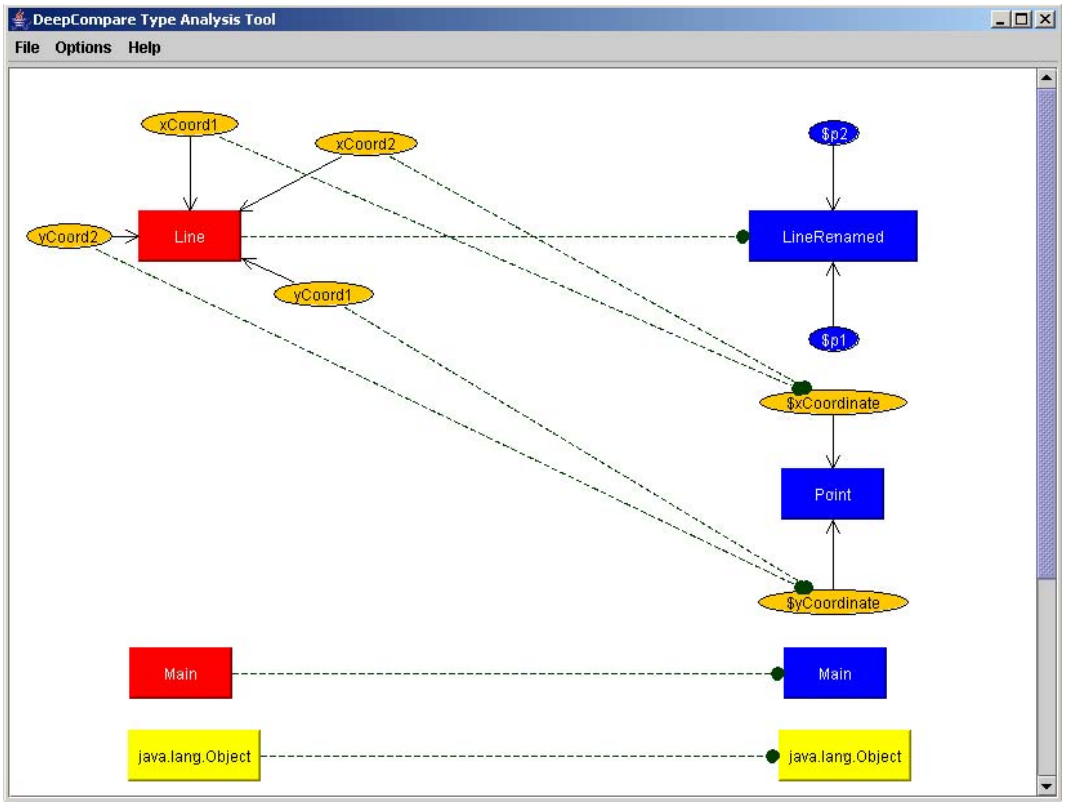


Fig. 2. DeepCompare at work. Additional detected similarities between JDK types are not shown for clarity.

two points of this line. In the new version, a new type (`Point`) was introduced and used by the Line object. In addition, the object was renamed. In this case, DeepCompare succesfully identified all equivalent types and members between two versions. In a first step, the types `Line` and `LineRenamed` were found to represent the same type, based on similar implemented functionality. A harvester further down the chain later detected the use of an encapsulation and succesfully associated the variables that represent the coordinates of the points that define our line.

While an example of this size hardly proves the general applicability of DeepCompare, experiments with larger projects have been executed and have shown promising results. A thorough description of a larger example is outside the scope of this position paper.

5 Future Work

Large portions of the Fresco methodology (e.g. the entire runtime platform: Draco and its dynamic update extension) are near completion. The design time tool DeepCompare is under very active development. Currently, the tool is functional with simple algorithms. In the near future, focus will be on the implementation of more complex harvesters in order to improve the recognition

of equivalent structures. The tool will be tested against larger projects and both success and failure scenarious will be investigated to further improve the tool. Future work also includes a full implementation of the code generator and extending it with an invariant checker. This will allow DeepCompare to verify the correctness of the proposed transformations. Future work also includes flagging possible problematic structures to the programmer (e.g. structures that would take too long to convert during a *live* update, complex conversions that require manual verification, ...).

6 Related Work

Most work in the field of live updates concentrates on the dynamic aspects. For procedural systems, Hicks ([9]) uses dynamic patches that consist of verifiable native code. With CONUS, the programmer can replace a component using a declarative specification of the desired changes ([11]). For object-oriented applications written in Java, systems were proposed that use a variety of methods in order to modify a running application. Technologies used include modifications to the Java Virtual Machine ([1,13]), language extensions ([4,8]) and meta-architectures ([16]). To give an exhaustive overview of existing systems is beyond the scope of this paper, and we refer to [9,18] for more complete surveys.

The problem of state transfer is usually left to the programmer, or ignored completely. Although certain systems include some tool support ([1,9]), this is often limited to the generation of a framework in which the user can implement the transition. The system developed by Hicks even automatically generates transfer code for variables whose names remain unchanged. In general however, little intelligent support is offered to transfer the state itself. For some systems, such as the delegation based approach by Kniesel (see [8]), state transfer is not relevant.

7 Conclusion

In this position paper we introduce the Fresco methodology for runtime evolution of component-oriented applications. At design time, components are instrumented with state transfer functionality. The developer is aided in this task by a tool: DeepCompare. This tool searches for equivalent data structures in two versions of a component and uses this information to partially generate the transition code. At runtime, a middleware platform takes care of all the details of the update itself.

References

[1] Andersson, J. and T. Ritzau, *Dynamic code update in* JDRUMS, in: *Workshop on Software Engineering for Wearable and Pervasive Computing*, 2000.

[2] Beck, K., "Extreme Programming Explained: Embrace Change," 1999.

[3] Berbers, Y., P. Rigole, S. V. Baelen and Y. Vandewoude, *Components and contracts in software development for embedded systems*, in: *Proc. of the first ECUMIS*, 2004, pp. 219–226.

[4] Costanza, P., *Dynamic object replacement and implementation-only classes*, in: *Proc. of Workshop on Component-Oriented Programming at ECOOP*, 2001.

[5] Demeyer, S., S. Ducasse and O. Nierstrasz, *Finding refactorings via change metrics*, in: *Proc. of OOPSLA*, 2000, pp. 166–177.

[6] Dockx, J., N. Smeets, K. Mertens and E. Steegmans, *jnome: A java meta-model in detail*, Technical Report CW323, KULeuven Department of Computerscience (2001).

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

[8] Günter Kniesel, *Type-safe delegation for run-time component adaptation*, in: *Proc. of ECOOP 99*, Lisbon, Portugal, 1999.

[9] Hicks, M., "Dynamic Software Updating," Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania (2001).

[10] Janssens, N., S. Michiels, T. Mahieu and P. Verbaeten, *Towards transparent hot-swapping support for producer-consumer components*, in: *Second Int. Workshop on Unanticipated Software Evolution*, Warshau, 2003, pp. 9–16.

[11] Kramer, J. and J. Magee, *The evolving philosophers problem: Dynamic change management*, IEEE Transactions on Software Engineering **16**, pp. 1293–1306.

[12] Lerner, B. S., *A model for compound type changes encountered in schema evolution*, ACM Transactions on Database Systems **25** (2000), pp. 83–127.

[13] Malabarba, S., R. Pandey, J. Gragg, E. Barr and J. F. Barnes, *Runtime support for type-safe dynamic java classes*, in: *Proc. of ECOOP*, 2000.

[14] Malpohl, G., J. Hunt and W. Tichy, *Renaming detection*, in: *Proc. of the 15th International Conference on Automated Software Engineering*, 2000.

[15] Rausch, A., *Software evolution in componentware - a practical approach*, in: *Proc. of the Australian Software Engineering Conference*, 2000.

[16] Redmond, B. and V. Cahill, *Iguana/j: Towards a dynamic and efficient reflective architecture for java.*, in: *Workshop on Reflection and Meta-Level Architectures at 14th European Conference on Object-Oriented Programming*, 2000.

[17] Urting, D., T. Holvoet, P. Rigole, Y. Vandewoude and Y. Berbers, *A tool for component based design of embedded software*, in: *Proc. of Tools Pacific 2002.*

[18] Vandewoude, Y. and Y. Berbers, *An overview and assessment of dynamic update methods for component-oriented embedded systems*, in: *Proc. of Software Engineering Research and Practice*, 2002, pp. 521–527.

[19] Vandewoude, Y. and Y. Berbers, *Supporting runtime evolution in seescoa*, Journal of Design & Process science **8** (2004), pp. 77–89.

[20] Vandewoude, Y., P. Rigole, D. Urting and Y. Berbers, *Draco : An adaptive runtime environment for components*, Technical Report CW372, Department of Computer Science, Katholieke Universiteit Leuven, Belgium (2003).

[21] Wise, M. J., *Improved detection of similarities in computer program and other texts*, in: *Twenty-Seventh SIGCSE Technical Symposium*, Philadelphia, 1996.

Dynamic software assembly for automatic deployment-oriented adaptation

Anthony Savidis

Institute of Computer Science, as@ics.forth.gr

Foundation for Research and Technology – Hellas (FORTH)

Abstract

The notion of software adaptation considered in this paper relates to the capability of making software systems adjustable to varying deployment requirements. In this context we seek for the necessary runtime infrastructure to allow software systems adapt on the fly to the particular execution requirements. The primary assumption is that the constituent components of a software system may have to be provided with alternative incarnations, each potentially addressing varying deployment needs. In this context, adaptation is treated as a runtime function of the system itself, realising a component selection and assembly process, since the deployment-specific parameters are only known upon execution start-up.

Key words: software adaptability, dynamic software assembly, deployment-oriented adaptation

1 Introduction

The need for software adaptability has been identified in [1], mainly emphasizing static software properties such as extensibility, flexibility and performance tunability, without negotiating the automatic and dynamic software assembly. Similarly, in [2], adaptability is also considered a key static property of software components, which can be pursued through aspectual decomposition, i.e., by employing aspect-oriented programming methods. In this paper, we are targeted in the engineering of software systems capable to dynamically activate alternative implementation versions of embedded software components through a runtime decision process, which relies on deployment-oriented decision parameters. To provide a more precise idea regarding dynamic software assembly based on deployment requirements, the application of the reported work in the context of dynamic User Interface assembly will be supplied. In this context, deployment parameters concerned individual user profiles, including abilities, preferences, expertise, etc. The dynamically assembled software artifacts concerned User Interface components.

In Figure 1, an excerpt from the Use Interface component structure of the AVANTI web browser [3] is shown; arrows indicate interface components whose activation and graphical embedding takes place at start-up conditionally, depending on the individual user profile (the deployment parameters for User Interfaces were

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

actually user profile parameters). The adaptation-oriented decision logic for the cases of Figure 1 where alternative implementations exist is provided in Figure 2.

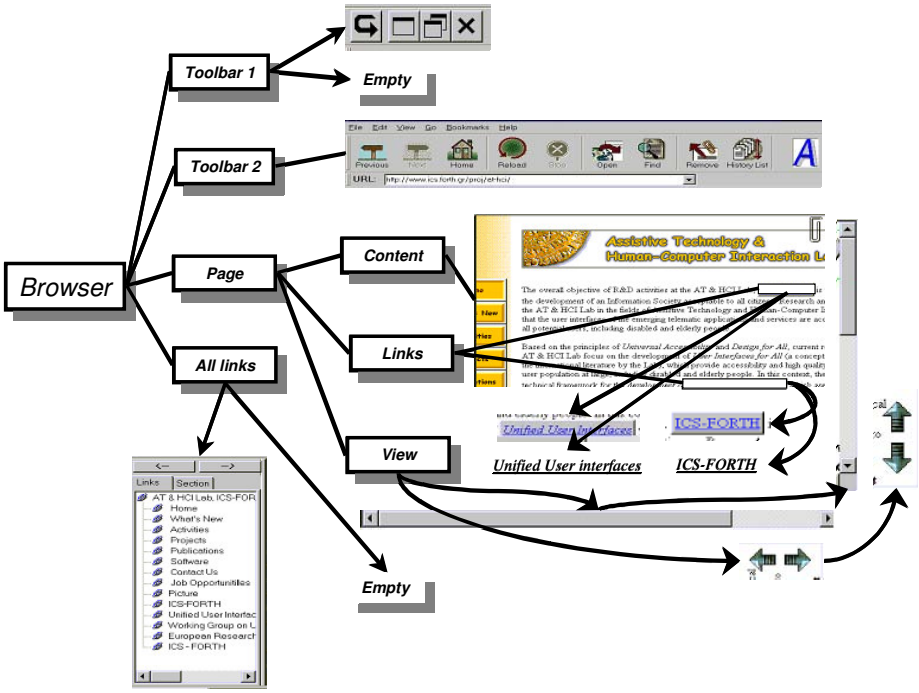


Figure 1: The hierarchical User Interface component structure of an adaptable browser; arrows indicate alternative implementations of components (empty indicates a component can be entirely omitted).

Component identifier	*Decision logic*

"Toolbar 1"	`if (user."CanUseUpperLimbs" == false) then`
	`  activate "ScanningToolbar";`
	`else`
	`  activate "Empty";`

"Links"	`if (user."WebExpertise" in {"Naïve", "Casual"}) then`
	`  activate "LinksAsButtons";`
	`else`
	`  activate "LinksAsUnderlinedText";`

"View"	`if (user."WebExpertise" in {"Naïve", "Casual"}) then`
	`  activate "EmbeddedScrollbars";`
	`else`
	`  activate "ScrollWindow";`

"All links"	`if (user."WebExpertise" in {"Naïve", "Casual"}) then`
	`  activate "LinkEnumeration";`
	`else`
	`  activate "Empty";`

Figure 2: The decision logic engaging user profile parameters (deployment profile) for adapted User Interface component selection and activation.

Following Figure 2, the decision logic engages user attributes (variables) within *if-then-else* rules that encompass activation statements. For example, *"Links"* is a component family with two alternative implementations, each associated uniquely with a descriptive identifier, e.g. *"LinksAsButtons"* and *"LinksAsUnderlinedText"*. This implies that the container of the *"Links"* family is capable to physically embed either of the alternatives, while the choice as to which *"Links"* instantiation is to be activated is taken by executing the decision logic upon application start-up. In this context, because of the fact that the *"Links"* component may have multiple alternative realisations it is called a *polymorphic* component, meaning it can be met in different

executions of the same interactive application with different forms. The software engineering approach for dynamic User Interface adaptation according to user profiles is extensively described in [4]. In this paper, the generalisation of dynamic User Interface assembly is reported, targeted in the implementation of software systems with the following properties: (a) they encapsulate alternative component implementations reflecting varying deployment requirements; (b) they are architecturally organized in ways enabling alternative implementations of polymorphic architectural components to be easily accommodated; (c) they encapsulate decision making driven by deployment parameter values; and (d) they perform a runtime software assembly process bringing together the necessary constituent components that bets-fit the particular deployment profile.

2 Architectural polymorphic decomposition

The key architectural implication due to the functional requirement for dynamic adaptation-oriented software assembly is the need for organization of implemented software components so as to enable dynamic architectural containment hierarchies. It should be noted that since containment concerns architectural decomposition relationships, i.e. components that logically encompass other components, containment always reflects a hierarchical structure. Other architectural views also exist, like dependency (or call) graphs, data exchange, etc., but those are not employed for the software assembly problem in our context. Overall, every software system has a hierarchical architectural view of its constituent components, which is actually of key importance when targeted in dynamic software assembly from runtime selected constituent components.

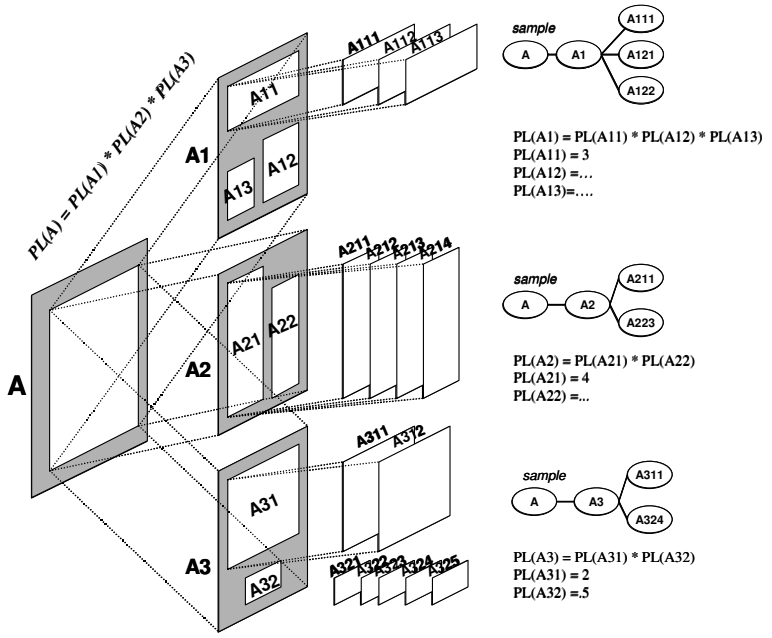


Figure 3: Illustration of polymorphic architectural containment for software components, showing the potential for multiple implementation instantiations of embedded components; it should be noted that the hierarchical architectural containment view is only of importance for dynamic software assembly.

In non-adaptable monomorphic software applications, developers typically program the hierarchical (containment) structure of architectural components through hard-coded associations that are determined during development time. However, in the context of adapted software delivery, the component containment hierarchies

should support two key features: (a) parent-child associations are always decided and applied during runtime; and (b) multiple alternatively candidate contained-instances are expected for composite components. The component organization method of dynamic polymorphic containment hierarchies is illustrated in the Figure 3. Following Figure 3, *PL* indicates the polymorphism factor, which provides the total number of all potential different run-time incarnations of a software component, recursively defined as the product of the polymorphic factors of constituent component classes. Practically, the actual number of plausible distinct software versions is less than *PL*, while it can be extracted by analysing the "diversity" of the deployment parameters. But since the deployment requirements may differentiate even per a component basis, the *PL* number does not only serve as a theoretical upper bound.

3 Dynamic assembly process

Since the hierarchical component-containment structure engages components, which can have alternative incarnations, it is implied that either the contained or the container components may vary. As a result, this hierarchical structure is not monomorphic, but reflects also a polymorphic discipline. In this context, the dynamic assembly process reflects the hierarchical traversal in the polymorphic containment hierarchy (see Figure 4), starting from the root component, to *decide*, *locate*, *instantiate* and *initiate* appropriately every target contained component. This process primarily concerns the architectural components that are actually *polymorphic*, i.e. architectural container components designed with alternative deployment-oriented decompositions.

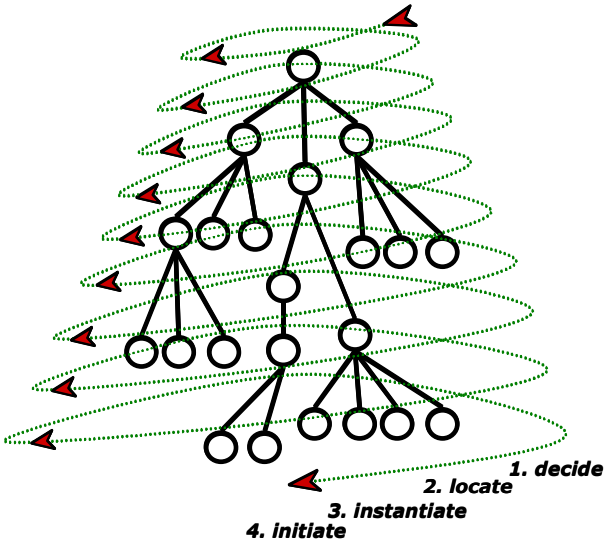


Figure 4: The traversal of the hierarchical containment architectural structure to: decide, locate, instantiate and initiate software components.

From the implementation point of view, the following software design decisions can be made:

- The containment-oriented architecture-component hierarchy has been implemented as a tree data structure, with polymorphic nodes triggering decision making sessions;
- Software components have been implemented as distinct independent software modules, implementing architecture-role generic Application Programming Interfaces (APIs), while exposing a singleton control-API for dynamic instantiation and name-based lookup;

- The software assembly procedure is actually carried out via two successive hierarchical passes:
  - Execution of decision sessions, to identify the specific selections for polymorphic architectural contexts, that will be part of the eventually delivered software;
  - Software assembly and start-up, through instantiation and initiation of all decided components.

Although the previous process is only conceptually illustrated under Figure 4, its implementation is quite straightforward when concerning component that are singleton classes: each alternative singleton derives from the basic base-class API (component-specific), while all singleton pointers are populated in a hash-map; the initial selection is simply made via name-based look-up. The implementation of dynamic assembly becomes much harder when polymorphic components concern normal program classes, instances of which are made explicitly via statements within the program source-code. More specifically:

- Let *A* be a programmer-defined class.
- *A a;* and *new A(…);* are two example statements for explicit instantiation of class *A* in the program source code.
- Let $A_1,\ldots,A_n$ be alternative deployment-oriented implementations of *A*; we need to allow instantiations to concern A_i, assuming A_i implementation is chosen upon decision making.
- We want to provide a generic instantiation style of the form: *A::Construct("A_i")*, thus supporting parameterization of the specific class name.

The previous required style signifies a departure from the traditional style of hard-coded class instantiations in the program source code to parameterized instantiations, enabling the class identifier to be supplied as a string argument. In other words, instead of *new Ai*, we want to support *A::Construct(A::GetDecidedClassId())*, where *GetDecidedClassId* is a static function returning the decided *A* version from $\{A_1,\ldots,A_n\}$. This represents a radically different perspective in class deployment, enabling orthogonal expansion of mutually exclusive class versions, while emphasizing deployment according to the base class API. The implementation of this technique is illustrated in the source code pattern of Figure 5. As shown in Figure 5, every class version implements a constructor functor class that is registered upon static class initialization in the class–specific dispatch table; this functor class, named *Constructor*, is responsible for dynamic class instantiation by calling the appropriate overloaded class constructor. This technique is a variation of double / dual dispatching. At the bottom of Figure 5, the parameterized deployment style is shown, with the traditional style of hard-coded class use put in comments. Clearly, the new style requires a little more code typing, however, it emphasizes far better deployment based on the basic class API, i.e. *Base**, while completely hiding the different class versions.

Additionally, it supports orthogonal extension of class versions, since the implementation of the dispatching method within the *Base* class is not dependent on derived classes; hence, once new derived classes are implemented according to the suggested pattern, those become automatically engaged in the adaptation process. This technique is easier to implement once classes become available over a component-ware technology, as they are already delivered over proxy APIs. Also, in cases of languages enabling dynamic class loading, like Java or Action Script, dynamic loading of class versions is straightforward.

```
class Base {
    public:
       struct CtorArgs_string {
              std::string arg;
              CtorArgs_string (const std::string& s) : arg(s){}
       };
       struct CtorArgs_int {
              int arg;
              CtorArgs_int (int i) : arg(i){}
       };
       struct CtorArgs_void  { CtorArgs_void (void){}       };

       class Ctor {
              public:
              virtual Base* operator()(CtorArgs_string&) = 0;
              virtual Base* operator()(CtorArgs_int&) = 0;
              virtual Base* operator()(CtorArgs_void&) = 0;
       };
       static const std::string GetDecidedClassId (void);

       template <class Args> static Base* Construct (Args& args) {
              std::map<std::string, Ctor*>::iterator i;
              i = ctorMap.find(GetDecidedClassId());
              assert(i != ctorMap.end());
              return (*i->second)(args);
       }
    protected:
       static std::map<std::string, Ctor*> ctorMap;
};

class Derived : public Base {
       class Constructor;
       friend class Constructor;
       Derived (const std::string&);
       Derived (int);
       Derived (void);
    public:
       class Constructor : public Base::Ctor {
              public:
              Base* operator()(CtorArgs_string& a)
                     { return new Derived(a.arg); }
              Base* operator()(CtorArgs_int& a)
                     { return new Derived(a.arg); }
              Base* operator()(CtorArgs_void& a)
                     { return new Derived; }
       };
       static void Initialise (void)
              { ctorMap["Derived"] = new Constructor; }
};

Base* b = Base::Construct(Base::CtorArgs_void());
// Derived* d = new Derived;
```

Figure 5: Implementing virtual destructors through runtime dispatching of class instantiations relying on class and argument type dispatching.

4 Key architectural ingredients

As it has been previously mentioned, the adopted notion of software adaptability reflects the functional properties of automatic software assembly, through decision-making that relies upon runtime software adaptation parameters. It should be noted that this is a fundamentally different target from formal methods related to software evolution, which focus on the automated transformation and evolution of software structures at development-time, according to diverse software requirements. The key architectural elements towards dynamic software assembly are:

- Hierarchical architectural view (component containment)
- Architectural context (sub-architecture that is subject to adaptation)
- Software component
- Software deployment parameters
- Software deployment scenarios
- Polymorphic architectural components
- Alternative encapsulated components
- Architectural decomposition
- Architectural role component indexing
- Architectural containment
- Functional-role abstraction APIs
- Mutually exclusive class versions

Those lead to an augmented vocabulary for the software architecture domain, mainly introducing the meta-elements necessary to accommodate runtime software assembly driven by decision-making for deployment adaptation, as illustrated in Figure 6.

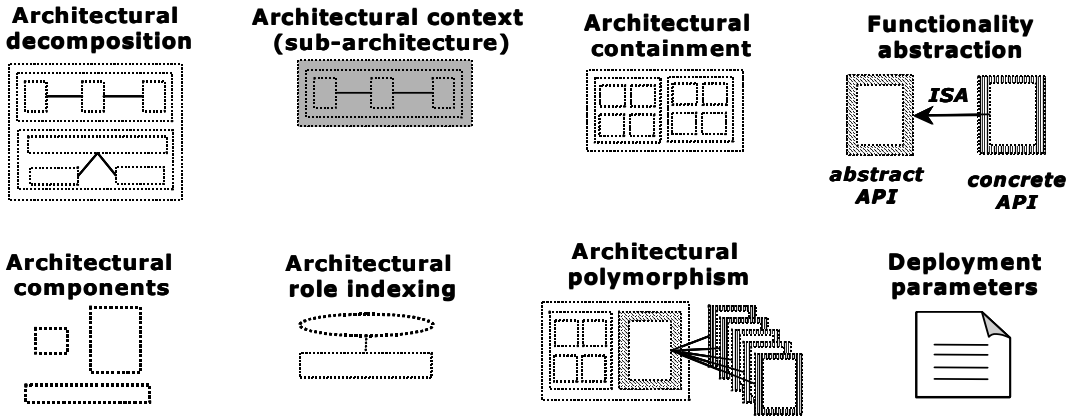


Figure 6: The key architecture meta-elements for dynamic software assembly, emphasizing the capability to accommodate alternative mutually exclusive implementation of components and classes within the same architecture.

5 Dynamic content delivery

In the context of the PALIO project [5], the software engineering method for dynamic software assembly has been effectively employed for adaptable information delivery over mobile devices to tourist users. The decision-making process was based on parameters such as nationality, age, location, interests or hobbies, time of day, visit history, and group information (i.e. family, friends, couple, colleagues, etc.). The information model reflected a typical relational database structure, while content retrieval was carried out using SQL queries in XML. In this context, in order to enable adapted information delivery, instead of implementing hard-coded SQL queries, query patterns have been designed, with specific polymorphic placeholders filled in by dynamically decided concrete sub-query patterns. For instance, as shown in Figure 7, particular data categories or even query operations may be left "open", with multiple alternatives, depending on runtime content-adaptation decision making.

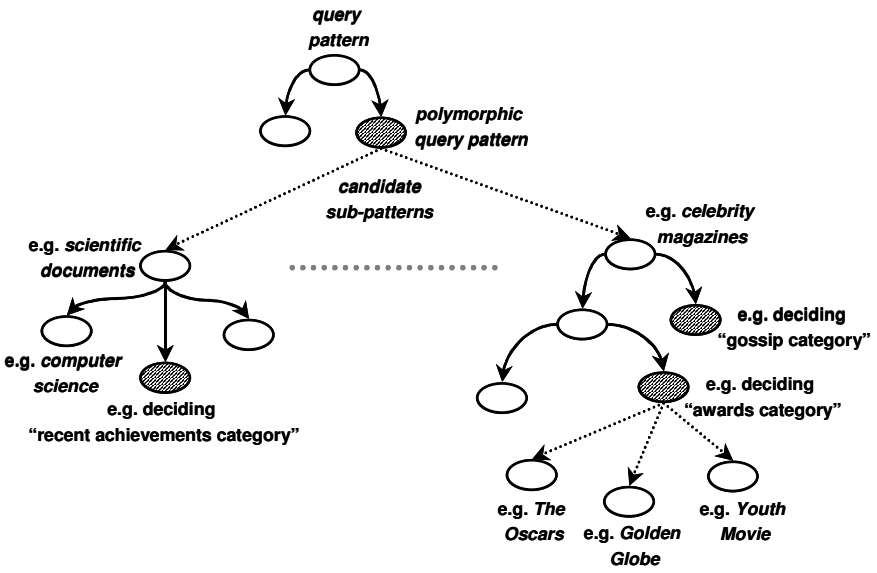


Figure 7: Polymorphic query patterns for adaptable query formulation.

6 Conclusions

This paper negotiates the software engineering of systems capable to realize a dynamic assembly behavior, from a pool of fully implemented software components, according to decision-making that is based on deployment-oriented requirements (in contrast to design-time decisions). During execution, the system reflects a runtime transformation behavior, in the sense that is capable to set-up itself on the fly according to the particular deployment requirements. To accomplish this behavior with design-time transformations, all plausible system versions, as combinations of the desirable components, need to be produced and delivered together. Clearly, this is an impractical method, while it does not allow the system to dynamically extend, e.g. by enabling downloading and installing new component versions addressing additional deployment needs.

7 References

[1] Fayad, M., Cline, M. (1996). Aspects of Software Adaptability. *In CACM Journal, 39 (10)*, October 1996, 58-59.

[2] Netinant, P., C. A. Constantinides, T. Elrad, M. E. Fayad. (200). Supporting Aspectual Decomposition in the Design ofAdaptable Operating Systems Using Aspect-Oriented Frameworks. Proceedings of 3rd *Workshop on Object-Orientation and Operating Systems* ECOOP-OOOWS, Sophia Antipolis, France, June 2000, pp. 36-46.

[3] Stephanidis, C., Paramythis, A., Sfyrakis, M., Savidis, A. (2001a). A Case Study in Unified User Interface Development: The AVANTI Web Browser. In *User Interfaces for All*, Stephanidis, C. (Ed), Lawrence Erlbaum, NJ, 525-568.

[4] Stephanidis, C., Savidis, A., & Akoumianakis, D. (2001b). Tutorial on "Universally accessible UIs: The Unified User Interface development". Tutorial in the ACM Conference on Human Factors in Computing Systems (CHI 2001), Seatle, Washington, 31 March - 5 April. Available at: http://www.ics.forth.gr/proj/at-hci/files/CHI_tutorial.pdf.

[5] Stephanidis, C., Paramythis, A., Zarikas, V., Savidis, A. (2004). The PALIO Framework for Adaptive Information Services. In A. Seffah & H. Javahery (Eds.), *Multiple User Interfaces: Cross-Platform Applications and Context-Aware Interfaces* (pp. 69-92). Chichester, UK: John Wiley & Sons, Ltd.