



3RD INTERNATIONAL WORKSHOP ON SOFTWARE EVOLUTION THROUGH TRANSFORMATIONS: EMBRACING CHANGE

PROCEEDINGS

Editors:

Jean-Marie Favré

Institut d'Informatique et Mathématiques Appliquées
Université Grenoble 1, France

Reiko Heckel

Department of Computer Science, University of Leicester, UK

Tom Mens

Software Engineering Lab, Université de Mons-Hainaut, Belgium

Natal - Brazil

September 17-22, 2006

Foreword

Since its birth as a discipline in the late 60ies Software Engineering had to cope with the breakdown of many of its original assumptions. Today we know that

- it is impossible to fix requirements up front;
- the design of the system is changing while it is being developed;
- the distinction between design time and run-time is increasingly blurred;
- a system's architecture will change or degrade while it is in use;
- technology will change more rapidly than it is possible to re-implement critical applications;

This recognition of lack of stability in software means that we have to cope with change, rather than defending against it. Processes, methods, languages, and tools have to be geared towards making change possible and cheap.

Transformations of development artifacts like specifications, designs, code, or run-time architectures are at the heart of many software engineering activities. Their systematic specification and implementation are the basis for a wide range of tools, from compilers and refactoring tools to model-driven CASE tools and formal verification environments. The workshop provides a forum for the discussion transformation-based techniques in software evolution.

Topics

Submissions to the workshop are based on a wide range of transformation formalisms like

- program transformation (over Java, C, or C++, etc.);
- model transformation (over UML and other visual languages);
- graph transformation;
- term rewriting;
- category theory, algebra, and logic;

discussing their application to software evolution activities like

- model-driven development;
- model and code refactoring, redesign and code optimisation;
- requirements evolution;
- reverse engineering, pattern detection, architecture recovery;
- architectural reconfiguration, self-organising or self-healing systems, service-oriented architectures;
- consistency management, co-evolution of models and code;
- merging of models, specifications, ontologies, etc;

Program Committee

The following program committee is responsible for the selection of papers.

- Luciano Baresi, Politecnico di Milano, Italy
- Thaís Batista, Federal University of Rio Grande do Norte, Brazil
- Paulo Borba, Universidade Federal de Pernambuco, Recife, Brazil
- Artur Boronat, Universidad Politécnica de Valencia, Spain
- Christiano de Oliveira Braga, Universidad Complutense de Madrid, Spain
- Andrea Corradini, Università di Pisa, Italy
- Mohammad El-Ramly, University of Leicester, UK
- Jean-Marie Favre, Université Grenoble 1, France [co-chair]
- Reiko Heckel, University of Leicester, UK [co-chair]
- Dirk Janssens, University of Antwerp, Belgium
- Tom Mens, Université de Mons-Hainaut, Belgium [co-chair]
- Anamaria Martins Moreira, Universidade Federal do Rio Grande do Norte, Natal, Brazil
- Leila Silva, Universidade Federal de Segipe, Brazil
- German Vega, Université Grenoble 1, France

Acknowledgement

The workshop is supported by the European Research Training Network *SegraVis* on Syntactic and Semantic Integration of Visual Modelling Techniques, the Integrated Project *Sensoria* on Software Engineering for Service-Oriented Overlay Computers, and the ERCIM Working Group on *Software Evolution*.

Contributed Papers

Optimizing Pattern Matching Compilation By Program Transformation, <i>Emilie Balland and Pierre-Etienne Moreau</i>	1
Towards Distributed BPEL Orchestrations, <i>Luciano Baresi, Andrea Maurino, and Stefano Modafferi</i>	15
EMF Model Refactoring based on Graph Transformation Concepts, <i>Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss</i>	29
Exogenous Model Merging by means of Model Management Operators, <i>Artur Boronat, Jose A. Carsi, and Isidro Ramos</i>	43
An Algorithm for Detecting and Removing Clones in Java Code, <i>Nicolas Juillerat and Beat Hirsbrunner</i>	63
Refactoring Information Systems: Handling Partial Compositions, <i>Michael Löwe, Harald König, Michael Peters, and Christoph Schulz</i>	75
An Approach to Invariant-based Program Refactoring, <i>Tiago Massoni, Rohit Gheyi, and Paulo Borba</i>	91
Generating Requirements Views: A Transformation-Driven Approach, <i>Lyrene Fernandes da Silva and Julio Cesar Sampaio do Prado Leite</i>	103
A MDE-Based Approach for Developing Multi-Agent Systems, <i>Viviane Silva, Beatriz de Maria, and Carlos Lucena</i>	115
From C++ Refactorings to Graph Transformations, <i>László Vidács, Martin Gogolla, and Rudolf Ferenc</i>	127

Selected contributions will appear in the Electronic Communications of EASST, the European Association of Software Science and Technology, under <http://tfs.cs.tu-berlin.de/ec-easst/>.



Optimizing Pattern Matching Compilation By Program Transformation

Emilie Balland *and Pierre-Etienne Moreau *

*UHP & LORIA, INRIA & LORIA,

email: {Emilie.Balland,Pierre-Etienne.Moreau}@loria.fr

***Abstract.** Motivated by the promotion of rewriting techniques and their use in major industrial applications, we have designed Tom: a pattern matching layer on top of conventional programming languages. The main originality is to support pattern matching against native data-structures like objects or records. While crucial to the efficient implementation of functional languages as well as rewrite rule based languages, in our case, this combination of algebraic constructs with arbitrary native data-structures makes the pattern matching algorithm more difficult to compile. In particular, well-known many-to-one automaton-based techniques cannot be used. We present a two-stages approach which first compiles pattern matching constructs in a naive way, and then optimize the resulting code by program transformation using rewriting. As a benefit, the compilation algorithm is simpler, easier to extend, and the resulting pattern matching code is almost as efficient as best known implementations.*

1 Introduction to Tom

Pattern matching is an elegant high-level construct which appears in many programming languages. Similarly to method dispatching in object oriented languages, it is essential in functional languages like Caml, Haskell, or SML. It is part of the main execution mechanism in rewrite rule based languages like ASF+SDF, ELAN, Maude, or Stratego.

In this paper, we present Tom¹ whose goal, similarly to Prop [9] or Pizza [11], is to integrate the notion of pattern matching into classical languages such as C and Java. Following the first ideas presented in [10], illustrated in Figure 1, a Tom program is a program written in a host language and extended by some new instructions like the `%match` construct. Therefore, a program can be seen as a list of Tom constructs interleaved with some sequences of characters. During the compilation process, all Tom constructs are dissolved and replaced by instructions of the host-language, as it is usually done by a preprocessor.

In order to understand the choices we have made when designing the pattern matching algorithm, it is important to consider Tom as a generic and *partial* compiler (like a pre-processor) which does not have any information about the host-language. In [2], Tom programs are described as islands anchored in

¹<http://tom.loria.fr>

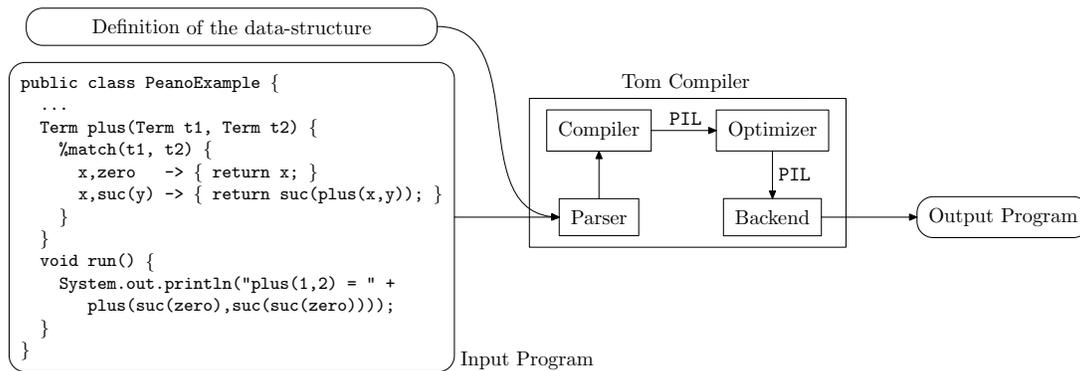


Figure 1: General architecture of Tom: the compiler generates an intermediate PIL program which is optimized before being pretty-printed by the back-end into the host-language.

host programs and the link between the two languages is formally defined in a generalized framework. In particular, the data-structure, against which the pattern matching is performed, *is not fixed*. In some sense, the data-structure is a parameter of the pattern matching, see [8] for more details. In practice, this means that a description of the data-structure (a mapping) has to be given to explain Tom how to access subterms for example.

In this paper, we present how the introduced pattern matching construct is compiled, using a program transformation approach. There exists several methods [4, 1, 6, 5] to efficiently compile pattern matching. The simplest ones, called *one-to-one*, inspect and compile each pattern independently. A more efficient approach consists in considering the system globally and building a discrimination network. These methods are called *many-to-one*, and they usually consist of three phases: constructing an automaton, optimizing it, and finally generating the implementation code. There are two main approaches to construct a matching automaton: one based on decision trees [4, 6] and the other on backtracking automata [1]. These two approaches emphasize the unavoidable compromise between speed and memory space [13].

In our case, we cannot assume that a function symbol (i.e. a node of a tree) is represented by an integer, like it is commonly done in other implementations of pattern matching. Therefore, the classical `switch/case` instruction can no longer be used to perform the discrimination. Since Tom supports several languages, it is also not possible to use an exception mechanism or a jump statement, like it can be done in ML compilers [7].

The approach chosen in Tom is to keep the optimization phase separated from the *one-to-one* compilation phase. This allows us to design algorithms which are generic, simpler to implement, easier to extend and maintain, and that can be formally certified [8]. In addition, this work allows to generate efficient implementations. In Section 2, we present the compilation algorithm and its intermediate language PIL. In Section 3, we introduce a set of rules which describes the optimizer and a strategy to efficiently apply them. In Section 4 we show that the optimizations are correct and improve the program in execution and size. Finally, in Section 5, some experimental results are given for several revealing examples. This



paper assumes some familiarity with term rewriting notations introduced in [3].

2 Compilation

To be data-structure independent and support several host-languages, Tom instructions, like `%match`, are compiled into an intermediate language code, called PIL, before being translated into the selected host-language. To compile the `%match` construct, we consider each rule independently.

Contrary to *many-to-one* algorithms which construct decision trees or pattern automata, given a pattern, it is traversed top-down and left-to-right. Nested if-then-else constructs are generated to ensure that constructors of the pattern effectively occur in the subject at a correct position. This technique is inefficient because, for a set of rules, identical tests may be repeatedly performed. The worst-case complexity is thus the product of the number of rules and the size of the subject.

The nested if-then-else are expressed in an intermediate language called PIL, whose syntax is given in Figure 2. Note that PIL has both functional and imperative flavors: the assignment instruction `let(variable, <term>, <instr>)` defines a scoped unmodifiable variable, whereas the sequence instruction `<instr>; <instr>` comes from imperative languages. A last particularity of PIL comes from the `hostcode(...)` instruction which is used to abstract part of code written in the underlying host-language. This instruction is parameterized by a list of PIL-variables which are used in this part of host-code.

PIL	::=	<code><instr></code>	<code><expr></code>	::=	<code>true false</code>
symbol	::=	<code>f ∈ ℱ</code>			<code>eq(<term>, <term>)</code>
variable	::=	<code>x ∈ ℳ</code>			<code>is_fsymb(<term>, symbol)</code>
<code><term></code>	::=	<code>t ∈ ℱ(ℱ, ℳ)</code>	<code><instr></code>	::=	<code>let(variable, <term>, <instr>)</code>
		<code> subterm_f(<term>, n)</code>			<code>if(<expr>, <instr>, <instr>)</code>
		<code>(f ∈ ℱ ∧ n ∈ ℕ)</code>			<code><instr>; <instr></code>
					<code>hostcode(variable*)</code>
					<code>nop</code>

Figure 2: PIL syntax

Similarly to functional programming languages, given a signature \mathcal{F} and a set of variables \mathcal{X} , the considered PIL language can directly handle *terms*, boolean values (`true`, `false`), and perform operations like checking that a given term t is rooted by a symbol f (`is_fsymb(t, f)`), accessing to the n -th child of a term t (`subtermf(t, n)`), or comparing two terms (`eq(t1, t2)`). The implementation of `subtermf`, `eq` and `is_fsymb` is given by the mapping which describes data-structures. To support the intuition, examples of Tom and PIL code are given in Figure 3.

We define PIL semantics as in [8] by a big-step semantics *à la* Kahn. To represent a substitution, we model an environment by a stack of assignments of terms to variables. The set of environments is noted \mathcal{Env} . The reduction relation of the big-step semantics is expressed on tuples $\langle \epsilon, \delta, i \rangle$ where ϵ is an environment, δ is a list of pairs (environment, host-code), and i is an instruction. Thanks to δ , we can keep track of the executed host-code blocks within their environment: the environment associated to each host-code construct gives the instances of all variables which appear in the block. A complete



<p>Tom code:</p> <pre> ...Java code %match(Term t) { f(a) => { print(...); } g(x) => { print(...x...); } f(b) => { print(...); } }Java code ... </pre>	<p>Generated PIL code:</p> <pre> hostcode(...); if(is_fsym(t,f), let(t1, subterm_f(t, 1), if(is_fsym(t1, a), hostcode(), nop)), nop); if(is_fsym(t,g), let(t1, subterm_g(t, 1), let(x, t1, hostcode(x))) nop); if(is_fsym(t,f), let(t1, subterm_f(t, 1), if(is_fsym(t1, b), hostcode(), nop)), nop); hostcode(...); </pre>
---	--

Figure 3: The left column shows a Tom program which contains three patterns: $f(a)$, $g(x)$, and $f(b)$, where x is a variable. As an example, when the second pattern matches t , this means that t is rooted by the symbol g , and the variable x is instantiated by its immediate subterm. The right column shows the corresponding PIL code generated by Tom. We can notice that this code is not optimal, but will hopefully be optimized by transformation rules afterwards.

definition of the semantics can be found in [3].

$$\langle \epsilon, \delta, i \rangle \mapsto_{bs} \delta', \text{ with } \epsilon \in \mathcal{Env}, \delta, \delta' \in [\mathcal{Env}, \langle instr \rangle]^*, \text{ and } i \in \langle instr \rangle$$

As PIL programs are predominantly constituted of if-then-else statements, the optimization rules will depend of the evaluation of expressions $e \in \langle expr \rangle$. In the following we introduce the notions of equivalence and incompatibility for expressions, and we consider two functions $eval$ and Φ . $eval$ is a function which given an environment ϵ and an expression e evaluates e in ϵ to obtain a value (i.e true for true or false for false). Given an environment ϵ and a host-code list δ , the evaluation of a program $\pi \in \text{PIL}$ results in a host-code list: $\langle \epsilon, \delta, \pi \rangle \mapsto_{bs} \delta'$. During this evaluation, expressions e , subterm of π , are evaluated in environments ϵ' . We call Φ the function that associates such an environment ϵ' to a sub-expression e of π : $\epsilon' = \Phi(\pi, e, \epsilon, \delta)$. More formal definitions can be found in [3].

Definition 1 Given a program π , two expressions e_1 and e_2 are said π -equivalent, and noted $e_1 \sim_{\pi} e_2$, if for all starting environment ϵ, δ , $eval(\epsilon_1, e_1) = eval(\epsilon_2, e_2)$ where $\epsilon_1 = \Phi(\pi, e_1, \epsilon, \delta)$ and $\epsilon_2 = \Phi(\pi, e_2, \epsilon, \delta)$.

Definition 2 Given a program π , two expressions e_1 and e_2 are said π -incompatible, and noted $e_1 \perp_{\pi} e_2$, if for all starting environment ϵ, δ , $eval(\epsilon_1, e_1) \neq eval(\epsilon_2, e_2)$ where $\epsilon_1 = \Phi(\pi, e_1, \epsilon, \delta)$ and $\epsilon_2 = \Phi(\pi, e_2, \epsilon, \delta)$.

We can now define two conditions which are sufficient to determine whether two expressions are π -equivalent or π -incompatible. Propositions 1 and 2 are interesting because the problem is generally undecidable [12], but here, conditions can be easily used in practice. Indeed `cond1` which ensures that



the two expressions are evaluated in the same environment is easy to be checked because of PIL language restrictions and `cond2` is a purely syntactic condition. Proofs of these propositions are in [3].

Proposition 1 *Given a program π and two expressions $e_1, e_2 \in \langle expr \rangle$ at different positions in, we have $e_1 \sim_{\pi} e_2$ if: $\forall \epsilon, \delta, \Phi(\pi, e_1, \epsilon, \delta) = \Phi(\pi, e_2, \epsilon, \delta)$ (**cond1**) and $e_1 = e_2$ (**cond2**).*

The equality = correspond to syntactic equality and the two considered expressions are in a different position in the program so the two environments of evaluation are not trivially equal.

Proposition 2 *Given a program π and two expressions $e_1, e_2 \in \langle expr \rangle$, we have $e_1 \perp_{\pi} e_2$ if: $\forall \epsilon, \delta, \Phi(\pi, e_1, \epsilon, \delta) = \Phi(\pi, e_2, \epsilon, \delta)$ (**cond1**) and `incompatible`(e_1, e_2) (**cond2**), where `incompatible` is defined as follows:*

$$\begin{array}{lcl} \text{incompatible}(e_1, e_2) & = & \text{match } e_1, e_2 \text{ with} \\ & | & \text{false, true} \quad \rightarrow \top \\ & | & \text{true, false} \quad \rightarrow \top \\ & | & \text{is_fsym}(t, f_1), \text{is_fsym}(t, f_2) \quad \rightarrow \top \quad \text{if } f_1 \neq f_2 \\ & | & \rightarrow - \quad \rightarrow \perp \end{array}$$

3 Optimization

An optimization is a transformation which reduces the size of code (*space optimization*) or the execution time (*time optimization*). In the case of PIL, the presented optimizations reduce the number of assignments (`let`) and tests (`if`) that are executed at run time. When manipulating abstract syntax trees, an optimization can easily be described by a rewriting system. Its application consists in rewriting an instruction into an equivalent one, using a conditional rewrite rule of the form $i_1 \rightarrow i_2 \text{ IF } c$.

Definition 3 *An optimization rule $i_1 \rightarrow i_2 \text{ IF } c$ rewrites a program π into a program π' if there exists a position ω and a substitution σ such that $\sigma(i_1) = \pi|_{\omega}$, $\pi' = \pi[\sigma(i_2)]_{\omega}$ and $\sigma(c)$ is verified. If $c = e_1 \sim e_2$ (resp. $c = e_1 \perp e_2$), we say that $\sigma(c)$ is verified when $\sigma(e_1) \sim_{\pi|_{\omega}} \sigma(e_2)$ (resp. $\sigma(e_1) \perp_{\pi|_{\omega}} \sigma(e_2)$).*

3.1 Reducing the number of assignments

This kind of optimization is standard, but useful to eliminate useless assignments. In the context of pattern matching, this improves the construction of substitutions, when a variable from the left-hand side is not used in the right-hand side for example.

3.1.1 Constant propagation.

This first optimization removes the assignment of a variable defined as a constant. Since no side-effect can occur in a PIL program, it is possible to replace all occurrences of the variable by the constant (written $i[v/t]$).

$$\text{ConstProp: } \text{let}(v, t, i) \rightarrow i[v/t] \text{ IF } t \in \mathcal{T}(\mathcal{F})$$



3.1.2 Dead variable elimination and Inlining.

Using a simple static analysis, these optimizations eliminate useless assignments:

$$\begin{aligned} \text{DeadVarElim: } \mathbf{let}(v, t, i) &\rightarrow i \text{ IF } \mathbf{use}(v, i) = 0 \\ \text{Inlining: } \mathbf{let}(v, t, i) &\rightarrow i[v/t] \text{ IF } \mathbf{use}(v, i) = 1 \end{aligned}$$

where $\mathbf{use}(v, i)$ is a function that computes an upper bound on the number of occurrences of a variable v in an instruction i .

3.1.3 Fusion.

The following rule merges two successive `let` which assign the same value to two different variables. This kind of optimization rarely applies on human written code, but in the context of pattern matching compilation (see Figure 3), this case often occurs. By merging the bodies, this allows to recursively perform some optimizations on subterms.

$$\text{LetFusion: } \mathbf{let}(v_1, t_1, i_1); \mathbf{let}(v_2, t_2, i_2) \rightarrow \mathbf{let}(v_1, t_2, i_1; i_2[v_2/v_1]) \text{ IF } t_1 \sim t_2$$

Note that the terms t_1 and t_2 must be compatible to ensure that values of v_1 and v_2 are the same at run time. We also suppose that $\mathbf{use}(v_1, i_2) = 0$. Otherwise, it would require to replace v_1 by a fresh variable in i_2 .

3.2 Reducing the number of tests

The key technique to optimize pattern matching consists in merging branches, and thus tests that correspond to patterns with identical prefix. Usually, the discrimination between branches is performed by a `switch/case` instruction. In `Tom`, since the data-structure is not fixed, we cannot assume that a symbol is represented by an integer, and thus, contrary to standard approaches, we have to use an `if` statement instead. This restriction prevents us from selecting a branch in constant time. The two following rules define the fusion and the interleaving of conditional blocks.

3.2.1 Fusion.

The fusion of two conditional adjacent blocks reduces the number of tests. This fusion is possible only when the two conditions are π -equivalent. Remind that the notion of π -equivalence means that the evaluation of the two conditions in a given program are the same (see Definition 1):

$$\text{IfFusion: } \mathbf{if}(c_1, i_1, i'_1); \mathbf{if}(c_2, i_2, i'_2) \rightarrow \mathbf{if}(c_1, i_1; i_2, i'_1; i'_2) \text{ IF } c_1 \sim c_2$$

To evaluate $c_1 \sim c_2$ (i.e. $c_1 \sim_{\pi} c_2$ with π the redex of the rule), we use Proposition 1. The condition $\Phi(\pi, \epsilon, \delta, e_1) = \Phi(\pi, \epsilon, \delta, e_2)$ (**cond1**) is trivially verified because the semantics of the sequence instruction preserves the environment ($\forall \delta, \epsilon, \Phi(\pi, \epsilon, \delta, i_1; i_2) = \Phi(\pi, \epsilon, \delta, i_1) = \Phi(\pi, \epsilon, \delta, i_2)$) and then $\forall \delta, \epsilon, \Phi(\pi, \epsilon, \delta, \sigma(c_1)) = \Phi(\pi, \epsilon, \delta, \sigma(c_2))$. We just have to verify that $e_1 = e_2$ (**cond2**), which is easy.



3.2.2 Interleaving.

As matching code consists of a sequence of conditional blocks, we would like to optimize blocks with π -incompatible conditions (see Definition 2). Some parts of the code cannot be both executed in a given environment, so swapping statically their order does not change the program behavior.

As we want to keep only one of the conditional block, we determine what instructions must be executed in case of success or failure of the condition and we obtain the two following transformation rules:

$$\begin{aligned} \text{if}(c_1, i_1, i'_1); \text{if}(c_2, i_2, i'_2) &\rightarrow \text{if}(c_1, i_1; i'_2, i'_1; \text{if}(c_2, i_2, i'_2)) \text{ IF } c_1 \perp c_2 \\ \text{if}(c_1, i_1, i'_1); \text{if}(c_2, i_2, i'_2) &\rightarrow \text{if}(c_2, i_2; i'_1, \text{if}(c_1, i_1, i'_1); i'_2) \text{ IF } c_1 \perp c_2 \end{aligned}$$

As for the equivalence in the `IfFusion` rule, to evaluate $c_1 \perp c_2$, we just have to verify that e_1 and e_2 are incompatible (`cond2`). Note that the two presented rules are not right-linear, therefore some code is duplicated (i'_2 in the first rule, and i'_1 in the second one). As we want to maintain linear the size of the code, we consider specialized instances of these rules with respectively i'_2 and i'_1 equal to `nop`.

$$\begin{aligned} \text{IfInterleaving: } \text{if}(c_1, i_1, i'_1); \text{if}(c_2, i_2, \text{nop}) &\rightarrow \text{if}(c_1, i_1, i'_1; \text{if}(c_2, i_2, \text{nop})) \text{ IF } c_1 \perp c_2 \\ \text{if}(c_1, i_1, \text{nop}); \text{if}(c_2, i_2, i'_2) &\rightarrow \text{if}(c_2, i_2, \text{if}(c_1, i_1, \text{nop}); i'_2) \text{ IF } c_1 \perp c_2 \end{aligned}$$

These two rules reduce the number of tests at run time because one of the tests is moved into the “else” branch of the other. In practice, we only use the first one labelled by `IfInterleaving`. The second rule can be instantiated and used to swap blocks. When i'_1 and i'_2 are reduced to the instruction `nop`, the second rule can be simplified into:

$$\text{if}(c_1, i_1, \text{nop}); \text{if}(c_2, i_2, \text{nop}) \rightarrow \text{if}(c_2, i_2, \text{if}(c_1, i_1, \text{nop})) \text{ IF } c_1 \perp c_2$$

As c_1 and c_2 are π -incompatible, we have the following equivalence:

$$\text{if}(c_2, i_2, \text{if}(c_1, i_1, \text{nop})) \equiv \text{if}(c_2, i_2, \text{nop}); \text{if}(c_1, i_1, \text{nop})$$

After all, we obtain the following rule corresponding to the swapping of two conditional adjacent blocks. This rule does not optimize the number of tests but is useful to bring closer blocks subject to be merged thanks to the strategy presented in the next section.

$$\text{IfSwapping: } \text{if}(c_1, i_1, \text{nop}); \text{if}(c_2, i_2, \text{nop}) \rightarrow \text{if}(c_2, i_2, \text{nop}); \text{if}(c_1, i_1, \text{nop}) \text{ IF } c_1 \perp c_2$$

3.3 Application strategy

From the rules presented in Section 3.1 and 3.2, we define a rewrite system. Without strategy, this system is clearly not confluent and not terminating. For example, the `IfSwapping` rule can be applied indefinitely because of the symmetry of incompatibility. The confluence of the system is not necessary as long as the programs obtained are semantically equivalent to the source program but the termination is an essential criterion. Moreover, the strategy should apply the rules to obtain a program as efficient as possible. Let us consider again the program given in Figure 3, and let us suppose that we interleave the last two patterns. This would result in the following sub-optimal program:



```

if(is_fsym(t, f), let(t1, subtermf(t, 1), if(is_fsym(t1, a), hostcode(), nop)), nop) ;
if(is_fsym(t, g), let(t1, subtermg(t, 1), let(x, t1, hostcode(x)))
    if(is_fsym(t, f), let(t1, subtermf(t, 1), if(is_fsym(t1, b), hostcode(), nop)), nop)

```

The *IfSwapping* and *IfFusion* rules can no longer be applied to share the $\text{is_fsym}(t, f)$ tests. This order of application is not optimal. As we want to grant *IfFusion*, the interleaving rule must be applied afterward, when no more optimization is possible.

The second matter is to ensure termination. The *IfSwapping* rule is the only rule that does not decrease the size or the number of assignments of a program. To limit its application to interesting cases, we define a condition which ensures that a swapping is performed only if it enables a fusion. This condition can be implemented in two ways, either in using a context, or in defining a total order on conditions noted $<$ (a lexicographic order for example). The second approach is more efficient: similarly to a swap-sort algorithm it ensures the termination of the algorithm. In this way, we obtain a new *IfSwapping* rule:

$$\text{if}(c_1, i_1, \text{nop}); \text{if}(c_2, i_2, \text{nop}) \rightarrow \text{if}(c_2, i_2, \text{nop}); \text{if}(c_1, i_1, \text{nop}) \text{ IF } c_1 \perp c_2 \wedge c_1 < c_2$$

Using basic strategy operators such as *Innermost*(s) (which applies s as many times as possible, starting from the leaves), $s_1 \mid s_2$ (which applies s_1 or s_2 indifferently), *repeat*(s) (which applies s as many times as possible, returning the last unailing result), and $s_1 ; s_2$ (which applies s_1 , and then s_2 if s_1 did not fail), we can define a strategy which describes how the considered rewrite system should be applied to normalize a PIL program:

```

Innermost(
  repeat(ConstProp | DeadVarElim | Inlining | LetFusion | IfFusion | IfSwapping) ;
  repeat(IfInterleaving)
)

```

Starting from the program given in Figure 3, we can apply the rule *IfSwapping*, followed by a step of *IfFusion*, and we obtain:

```

if(is_fsym(t, f), let(t1, subtermf(t, 1), if(is_fsym(t1, a), hostcode(), nop))
    ; let(t1, subtermf(t, 1), if(is_fsym(t1, b), hostcode(), nop)), nop) ;
if(is_fsym(t, g), let(t1, subtermg(t, 1), let(x, t1, hostcode(x))), nop)

```

Then, we can apply a step of *Inlining* to remove the second instance of t_1 , a step of *LetFusion*, and a step of *Interleaving* ($\text{is_fsym}(t_1, a)$ and $\text{is_fsym}(t_1, b)$ are π -incompatible). This results in the following program:

```

if(is_fsym(t, f), let(t1, subtermf(t, 1),
    if(is_fsym(t1, a), hostcode(), if(is_fsym(t1, b), hostcode(), nop))), nop) ;
if(is_fsym(t, g), let(x, subtermg(t, 1), hostcode(x)), nop)

```

Since $\text{is_fsym}(t, f)$ and $\text{is_fsym}(t, g)$ are π -incompatible, we can apply a step of *IfInterleaving*, and get the irreducible following program:



```

if(is_fsym(t, f),
  let(t1, subtermf(t, 1), if(is_fsym(t1, a), hostcode(), if(is_fsym(t1, b), hostcode(), nop))),
  if(is_fsym(t, g), let(x, subtermg(t, 1), hostcode(x)), nop)

```

4 Properties

When performing optimization by program transformation, it is important to ensure that the generated code has some expected properties. The use of formal methods to describe our optimization algorithm allows us to give proofs. In this section we show that each transformation rule is correct, in the sense that the the optimized program has the same observational behavior as the original. We also show that the optimized code is both more efficient, and smaller than the initial program.

4.1 Correction

Definition 4 Given π_1 and π_2 two well-formed PIL programs, they are semantically equivalent, noted $\pi_1 \sim \pi_2$, when:

$$\forall \epsilon, \delta, \exists \delta' \text{ s.t. } \langle \epsilon, \delta, \pi_1 \rangle \mapsto_{bs} \delta' \text{ and } \langle \epsilon, \delta, \pi_2 \rangle \mapsto_{bs} \delta'$$

Definition 5 A transformation rule r is correct if for all well-formed program π , r rewrites π in π' (Definition 3) implies that $\pi \sim \pi'$ (Definition 4).

From this definition, we prove that every rule given in Section 3 is correct. For that, two conditions have to be verified:

1. π' is well-formed,
2. $\forall \epsilon, \delta$, the derivations of π and π' lead to the same result δ' .

The first condition is quite easy to verify. The second one is more interesting: we consider a program π , a rule $l \rightarrow r$ IF c , a position ω , and a substitution σ such that $\sigma(l) = \pi|_{\omega}$. We have $\pi' = \pi[\sigma(r)]_{\omega}$. We have to compare the derivations of π and π' in the context ϵ, δ .

- when ω is the empty position (which corresponds to the root), we have to compare the derivation tree of $\pi = \sigma(l)$ and $\pi' = \sigma(r)$,
- otherwise, we consider the derivation of π (resp. π'): there is a step which needs in premise the derivation of $\pi|_{\omega}$ (resp. $\pi[\sigma(r)]_{\omega}$). This is the only difference between the two trees.

In both cases, we have to verify that $\pi|_{\omega} = \sigma(l)$ and $\sigma(r)$ have the same derivation in a given context:

- equal to ϵ, δ when ω is the empty position,



- otherwise, we have to consider the instruction i which immediately contains $\sigma(l)$ (resp. $\sigma(r)$). The context is defined by the context in which i is evaluated in the derivation tree of π (resp. π').

In the following, we give one representative proof of correction: `IfSwapping`². To simplify the proof we consider l, r and c instead of $\sigma(l), \sigma(r)$ and $\sigma(c)$. In this rule, $l = \text{if}(c_1, i_1, \text{nop}); \text{if}(c_2, i_2, \text{nop})$ and $r = \text{if}(c_2, i_2, \text{nop}); \text{if}(c_1, i_1, \text{nop})$. To prove that $\pi \sim \pi'$, we have to verify that for a given ϵ, δ, l and r have the same derivation. Since c_1 and c_2 are π -incompatible, three cases have to be studied:

Case 1: $\text{eval}(\epsilon, c_1) = \text{true}$ and $\text{eval}(\epsilon, c_2) = \text{false}$

$$\frac{\frac{\langle \epsilon, \delta, i_1 \rangle \mapsto_{bs} \delta' \quad \text{eval}(\epsilon, c_1) = \text{true}}{\langle \epsilon, \delta, \text{if}(c_1, i_1, \text{nop}) \rangle \mapsto_{bs} \delta'} \quad \frac{\langle \epsilon, \delta', \text{nop} \rangle \mapsto_{bs} \delta' \quad \text{eval}(\epsilon, c_2) = \text{false}}{\langle \epsilon, \delta', \text{if}(c_2, i_2, \text{nop}) \rangle \mapsto_{bs} \delta'}}{\langle \epsilon, \delta, \text{if}(c_1, i_1, \text{nop}); \text{if}(c_2, i_2, \text{nop}) \rangle \mapsto_{bs} \delta'}$$

We now consider the program $\text{if}(c_2, i_2, \text{nop}); \text{if}(c_1, i_1, \text{nop})$.

Starting from the same environment ϵ and δ , we show that the derivation leads to the same state δ' , and thus prove that $\text{if}(c_1, i_1, \text{nop}); \text{if}(c_2, i_2, \text{nop})$ and $\text{if}(c_2, i_2, \text{nop}); \text{if}(c_1, i_1, \text{nop})$ are equivalent:

$$\frac{\frac{\langle \epsilon, \delta, \text{nop} \rangle \mapsto_{bs} \delta \quad \text{eval}(\epsilon, c_2) = \text{false}}{\langle \epsilon, \delta, \text{if}(c_2, i_2, \text{nop}) \rangle \mapsto_{bs} \delta} \quad \frac{\langle \epsilon, \delta, i_1 \rangle \mapsto_{bs} \delta' \quad \text{eval}(\epsilon, c_1) = \text{true}}{\langle \epsilon, \delta, \text{if}(c_1, i_1, \text{nop}) \rangle \mapsto_{bs} \delta'}}{\langle \epsilon, \delta, \text{if}(c_2, i_2, \text{nop}); \text{if}(c_1, i_1, \text{nop}) \rangle \mapsto_{bs} \delta'}$$

Since π and π' are well-formed, their derivation in a given context are unique (see [3]). $\langle \epsilon, \delta, i_1 \rangle \mapsto_{bs} \delta'$ is part of these derivation trees, so it is unique, and δ' is identical in both derivations.

Case 2: $\text{eval}(\epsilon, c_1) = \text{false}$ and $\text{eval}(\epsilon, c_2) = \text{true}$, the proof is similar.

Case 3: $\text{eval}(\epsilon, c_1) = \text{false}$ and $\text{eval}(\epsilon, c_2) = \text{false}$, the proof is similar.

4.2 Time and space reduction

To show that the optimized code is both more efficient, and smaller than the initial program, we consider two measures:

- the *size* of a program π is the number of instructions which constitute the program,
- the *efficiency* of a program π is determined by the number of tests and assignments which are performed at run time.

²The other proofs can be found in [3]



It is quite easy to verify that each transformation rule does not increase the size of the program: `DeadVarElim`, `ConstProp`, `Inlining`, and `LetFusion` decrease the size of a program, whereas `IfFusion`, `IfInterleaving` and `IfSwapping` maintain the size of the transformed program.

It is also clear that no transformation can reduce the efficiency of a given program:

- each application of `DeadVarElim`, `ConstProp`, and `Inlining` reduces by one the number of assignment that can be performed at run time,
- `IfFusion` reduces by one the number of tests,
- `IfInterleaving` also decreases the number of tests when the first alternative is chosen. Otherwise, there is no optimization,
- `IfSwapping` does not modify the efficiency of a program.

The program transformation presented in Section 3 is an optimization which improves the efficiency of a given program, without increasing its size. Similarly to [5], this result is interesting since it allows to generate efficient pattern matching implementations whose size is linear in the number and size of patterns.

5 Experimental Results

The Tom compiler is written in Tom and Java. Therefore, the presented algorithm described using rules and strategies, has been implemented in Tom. As illustrated Figure 1, the optimizer is just an extra phase of the compiler, which is now integrated into the main distribution using the strategy given in Section 3.3. In order to illustrate the efficiency of the compiler we have selected several representative programs and measured the effect of optimization in practice:

	Fibonacci	Eratosthene	Langton	Gomoku	Nspk	Structure
Tom Java	21.3 s	174.0 s	15.7 s	70.0 s	1.7 s	12.3 s
Tom Java Optimized	20.0 s	2.8 s	1.4 s	30.4 s	1.2 s	11.3 s

- `Fibonacci` computes 500 times the 18th Fibonacci number, using a Peano representation. On this example, the optimizer has a small impact because the time spent in matching is smaller than the time spent in allocating successors and managing the memory.

- `Eratosthene` computes prime numbers up to 1000, using associative list matching. The improvement comes from the `Inlining` rules which avoids computing a substitution unless the rule applies (i.e. the conditions are verified).

- `Langton` is a program which computes the 1000th iteration of a cellular automaton, using pattern matching to implement the transition function. This example is interesting because it contains more than 100 (ground) patterns. Starting from a simple one-to-one pattern matching algorithm, the optimizer performs program transformations such that a pair (position,symbol) is never tested more than once. This



interesting property, which characterizes deterministic automata based approaches, can unfortunately not be generalized to any program.

- Gomoku looks for five pawn on a go board, using list matching. This example contains more than 40 patterns and illustrates the interest of test-sharing.
- Nspk implements the verification of the Needham-Schroeder Public-Key Protocol.
- Structure is a prover for the Calculus of Structures where the inference is performed by pattern matching and rewriting.

The following table gives some comparisons with other well known implementations.

	Fibonacci	Eratosthene	Langton
Tom Java Optimized	20.0 s	2.8 s	1.4 s
Tom C Optimized	0.95 s	0.36 s	0.84 s
OCaml	0.44 s	0.7 s	1.36 s
ELAN	0.77 s	0.8 s	1.26 s

All these examples are available on the Tom web page. The measures have been done on a PowerMac 2 GHz, using Java 1.4.2, gcc 4.0, and Ocaml 3.09. They show that the proposed approach is effective in practice and allows Tom to become competitive with state of the art implementations such as OCaml. We should remind that Tom is not dedicated to a unique language. In particular, the fact that data-structure can be user-defined contrary to functional languages prevents us from using exception, goto, and switch constructs and thus optimizations like those presented in [5].

6 Conclusion

In this paper, we have presented a new approach to compile pattern matching. This method is based on well-attested program optimization methods. Separating compilation and optimization in order to keep modularity, and to facilitate extensions is long-established in the compiler construction community. Using a program transformation and a formal method approach is an elegant way to describe, implement, and certify the proposed optimizations. This work is closed to Sestoft approach [14] which compiles naively ML-style pattern matches and by partial evaluation removes redundant cases instead of constructing directly the decision tree. Moreover, this two-stage pattern compilation is directly implemented in Tom and shows how Tom language is well-adapted for program analysis-transformation.

We have only be interested in optimizing syntactic matching and thus considered a subset of PIL language. As Tom already manages associativity, a future work will consist in developing new transformation rules adapted to this theory, without having to change the rules relative to syntactic one. However, note that the presented rules remain correct when considering an extension of PIL.

This paper shows that using program transformation rules to optimize pattern matching is an efficient solution, with respect to algorithms based on automata. The implementation of this work combined with the formal validation of pattern matching [8] is another step towards the construction of certified/certifying optimizing compilers.



References

- [1] Lennart Augustsson. Compiling pattern matching. In *Proceedings of the conference on Functional Programming Languages and Computer Architecture*, pages 368–381. Springer-Verlag, 1985.
- [2] Emilie Balland, Claude Kirchner, and Pierre-Etienne Moreau. Formal islands. In M. Johnson and V. Vene, editors, *Proceedings of the 11th international conference on Algebraic Methodology and Software Technology*, volume 4019 of *LNCS*, pages 51–65. Springer-Verlag, 2006.
- [3] Emilie Balland and Pierre-Etienne Moreau. Optimizing pattern matching by program transformation. Technical report, INRIA-LORIA, 2005. <http://hal.inria.fr/inria-00000763>.
- [4] Luca Cardelli. Compiling a functional language. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 208–217, 1984.
- [5] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *Proceedings of the sixth International Conference on Functional Programming*, pages 26–37. ACM Press, 2001.
- [6] Albert Gräf. Left-to-right tree pattern matching. In *Proceedings of the 4th international conference on Rewriting Techniques and Applications*, volume 488 of *LNCS*, pages 323–334. Springer-Verlag, 1991.
- [7] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.
- [8] Claude Kirchner, Pierre-Etienne Moreau, and Antoine Reilles. Formal validation of pattern matching code. In P. Barahone and A. Felty, editors, *Proceedings of the 7th international conference on Principles and Practice of Declarative Programming*, pages 187–197. ACM, July 2005.
- [9] Allen Leung. C++-based pattern matching language, 1996.
- [10] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.
- [11] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 146–159. ACM Press, USA, 1997.
- [12] Oliver R uthing, Jens Knoop, and Bernhard Steffen. Detecting equalities of variables: Combining efficiency with precision. In A. Cortesi and G. Fil e, editors, *SAS*, volume 1694 of *LNCS*, pages 232–247. Springer-Verlag, 1999.
- [13] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. *SIAM Journal on Computing*, 24(6):1207–1234, 1995.



- [14] Peter Sestoft. ML pattern match compilation and partial evaluation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *LNCS*, pages 446–464. Springer-Verlag, 1996.



Towards Distributed BPEL Orchestrations

Luciano Baresi *, Andrea Maurino **, and Stefano Modafferi *

*Dipartimento di Elettronica e Informazione
Politecnico di Milano

Piazza L. da Vinci 32, I-20133 Milano, Italy

**Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano - Bicocca
Via Bicocca degli Arcimboldi 8, I-20126 - Milano, Italy

***Abstract.** Web services are imposing as the technology to integrate highly heterogeneous systems. BPEL, the standard technology to compose services, assumes a single "orchestrator" that controls the execution flow and coordinates the interactions with selected services. This centralized approach simplifies the coordination among components, but it is also a too heavy constraint. To this end, the paper introduces the idea of distributed orchestrations and presents a proposal to couple BPEL and distributed execution in mobile settings. The approach—exemplified on a simple case study—transforms a centralized BPEL process into a set of coordinated processes. An explicit meta-model and graph transformation supply the formal grounding to obtain a set of related processes, and to add the communication infrastructure among the newly created processes. The paper also presents a communication infrastructure based on tuple spaces to make the different orchestrators interact in mobile contexts.*

Keywords: WS-BPEL, Graph transformation, Distributed system

1 Introduction

Web services are a well-known technology to implement and deploy *service-oriented* systems. Their flexibility is playing a key role to integrate highly heterogeneous systems. Specific technologies are hidden behind simple XML interfaces written in WSDL (Web Services Description Language [CCMW01]) and services interact by means of SOAP (Simple Object Access Protocol) messages—another XML-based technology. The same ease and freedom do not apply to the paradigms with which services can be composed to support business processes. WS-BPEL (Web Services Business Process Execution Language [BEA03]), the most prominent standard technology to compose Web services, only allows for a "strict" centralized approach to composition, where a single *orchestrator* controls the execution flow and



coordinates the interactions with selected services.

This centralized approach to execution helps simplify the coordination among components, but it is also a too heavy constraint for those systems that have a distributed nature and imply the cooperation of decentralized (maybe nomadic) actors. The Web services community is already pushing more decentralized approaches to composition. Choreographies (WS-CDL, Web Services Choreography Description Language [Nic04]) —instead of orchestrations— support peer-to-peer interactions among services, and event-based architectures, with WS-Notification [Aka04] as one of the emerging standards in this direction, support even looser cooperations among Web services, where the interaction is limited to reacting to events generated within the composition. Unfortunately, available technology is still limited and it is not as widely accepted as WS-BPEL.

Even if we stick to a workflow-like view of the Web services composition, in many cases, the centralized process needs to be deployed in a distributed setting. The "conceptual" monolithic process must be partitioned into a well-organized set of sub-processes, and the implied coordination and synchronization actions among the parts must be suitably designed and implemented. This is the case, for example, of cross-departments business processes, where each division is in charge of a particular fragment of the process, and they all concur to the success of the global process. Nowadays, we can also envisage another, completely different, scenario with the centralized model split because it has to be executed by a set of federated mobile devices. No single device has the capability of executing the whole process, but they all can contribute with the execution of a dedicated portion. The scenario might become even bolder if we thought of moving the partitioning process at runtime to dynamically select and allocate the pieces of a process to the devices that are available, and have enough resources, at a given moment. Nomadic users are supposed to move with their devices, and thus the set of cooperating devices can change. This scenario could be useful for disaster recovery situations, and more in general in all those cases where a stable and reliable network is not available, but the process must rely on a mobile ad-hoc network.

Given the wide diffusion of WS-BPEL, and the good set of supporting tools, the paper tackles the problem of decentralizing the execution of WS-BPEL processes by fostering the idea of *decentralized orchestration*. It proposes an automatic and formal approach to partition WS-BPEL processes, and to add the necessary synchronization and coordination stuff. The approach transforms a centralized WS-BPEL process into a set of coordinated processes. An explicit and precise meta-model [Obj02] and graph transformation systems [BH02] supply the formal grounding to slice a single WS-BPEL process, obtain a set of related processes, and add the communication infrastructure among the newly created processes to both exchange data and propagate the execution flow. The paper demonstrates the approach on a simple case study.

The approach is supported by two different families of tools, which are currently under development. We are working on a CASE tool, based on AGG (Attributed Graph Grammar [Bey92]), that implements *partitioning rules* and automatize the whole "slicing" process. We are also working on the infrastructure to execute such federated WS-BPEL processes.

The results presented in this paper are an evolution of those presented in [BMM04]. More specifically, the novel contributions of this paper are: (1) an explicit meta-model for WS-BPEL specifications, (2) partitioning rules that address the information exchange among cooperating processes, (3) partitioning rules for fault, compensation, and event handlers, and (4) a more complete example application.

The rest of the paper is organized as follows. Section 2 introduces the formal ingredients, that is, the

meta-model and partitioning rules. Section 3 discusses the approach on an example application. Section 4 surveys related approaches, and Section 5 concludes the paper.

2 Partitioning rules

This section introduces the approach to transform a WS-BPEL model into a set of federated models. The first part of the section introduces the meta-model, specified to render the key elements of WS-BPEL, and graph transformation systems. The second part presents *partitioning rules*. It describes one rule thoroughly, and concentrates on the key aspects for the others. Lack of space does not allow us to present all rules in detail, but interested readers can refer to [BMM05] for the whole graph transformation system.

2.1 Meta-model

Even if WS-BPEL is a complex workflow language for describing Web services compositions, Figure 1 only comprises those elements that are used by our approach, that is, that are handled within partitioning rules. The meta-model presented here borrows some concepts from the work presented in [Ga03], but we explicitly decided to get rid of all those elements that, even if are part of the WS-BPEL specification, and thus should be part of a complete meta-model, are not considered by the partitioning process.

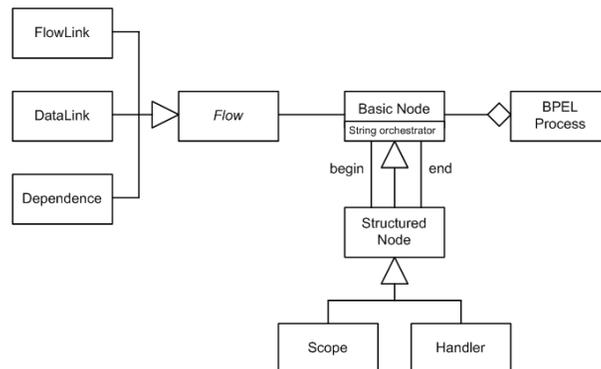


Figure 1: Dedicated WS-BPEL metamodel

Straightforwardly, a `BPEL Process` comprises several `Basic Nodes`. This class is the key component of the whole meta-model. `Basic Nodes` correspond to the basic activities that can be performed by a WS-BPEL process, like `invoke` and `receive`. `Structured Nodes`, which are a specialization of the first ones, correspond to the typical constructs of workflow languages supposed by WS-BPEL, like `switch`, `flow`, `sequence`, and `pick`. We treat each `Structured Node` by means of two `Basic Nodes` to identify the first and the last element of the composed statement. `Scopes` and `Handlers` are seen as special-purpose `Structured Nodes` since they both embed a set of `Basic`



Nodes. We do not use containment relations to render the embedding of some nodes into others, but we exploit associations `begin` and `end` to identify the scope of each structured construct.

Nodes are characterized by the `orchestrator` that is in charge of their execution and are connected through `Flows`. These, in turn, can be `FlowLinks`, to render order of execution among nodes, `DataLinks`, to define data dependencies among nodes, and `Dependences`, to relate the pairs of `invoke/receive Basic Nodes` added to synchronize components that have to be executed by two different executors (orchestrators).

This is not a "complete" and constrained meta-model. This means that we cannot be used to probe the consistency of WS-BPEL specifications. The meta-model is simple and "unprecise" since we assume that initial WS-BPEL models are correct and that the partitioning rules, that is, the graph transformation rules, only produce correct models¹.

2.2 Partitioning rules

Partitioning rules must be precise enough to allow for correct and automatic transformations and are supposed to work on the graph structure (i.e., an instance of the meta-model) that is behind any WS-BPEL process. These two requirements led us to consider graph transformation systems [BH02], along with the tool support they offer, as the right means to specify them.

Partitioning rules are introduced here incrementally. We start introducing the rules to control the execution flow, already presented in [BMM04], then we move to those that oversee data exchange and manage handlers. Since all rules are "similar", here we only describe one rule in detail and we invite readers to refer to [BMM05] for the complete set. All the other rules are presented by concentrating on the problems behind them, instead of showing how the different instances of meta-model elements are created, deleted, and modified.

In this paper, we introduce *partitioning rules* for controlling the execution flow by means of the example of Figure 2, which shows the meta-model of a simple WS-BPEL `Switch` statement with a `Basic Node` in each branch. The figure also shows that different nodes are associated with different orchestrators: this means that the designer has already decided how to split the process, that is, how to allocate the different sub-processes on available executors.

To partition the simple example process of Figure 2, we start by setting links to state the dependence, in terms of execution flow, between two WS-BPEL activities that are controlled by different orchestrators. Figure 3 shows the rule. The left-hand side describes when the rule can be applied, while the right-hand side shows the final result. The rule starts with two nodes that are associated with different orchestrators and introduces two new `Basic Nodes` to synchronize the execution of the first part with the execution of the second part. These two nodes are a pair of `invoke/receive` activities to forbid the second activity to start before the completion of the first one.

The partitioning process continues by identifying all the other pairs of nodes that must be synchronized and by adding special-purpose `Basic Nodes` to notify the branch followed by the `Switch` to all involved orchestrators.

¹This is not explicitly demonstrated in this paper, but readers can refer to [BMM04] for a first informal demonstration limited to the rules that deal with the execution flow.

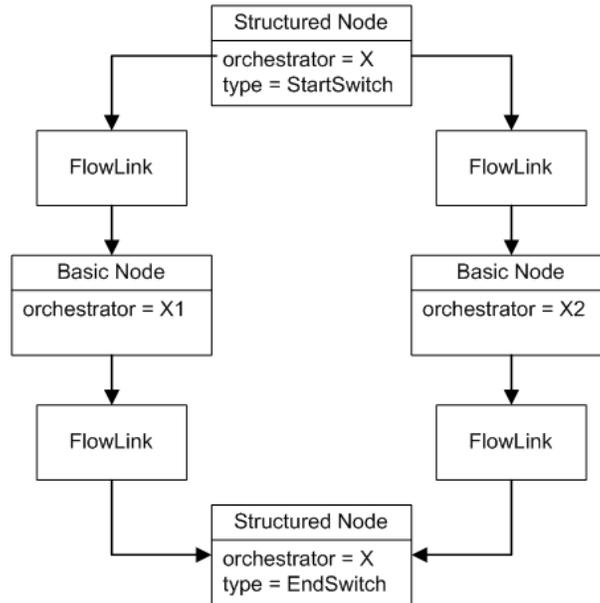


Figure 2: Example meta-model of a simple Switch construct

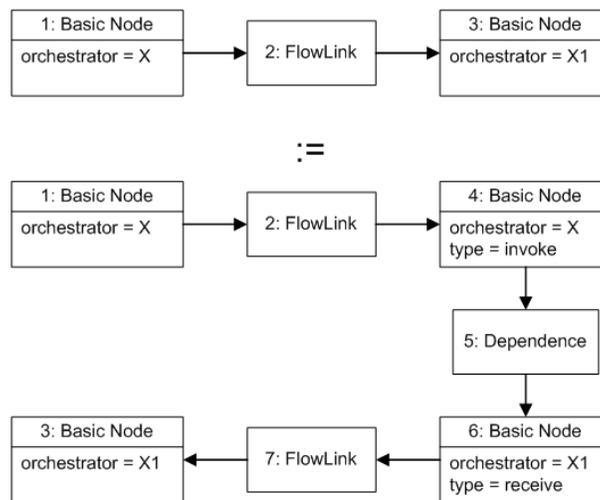


Figure 3: Rule AddDependence



The partitioning process also takes into account data distribution. According to the WS-BPEL specification, the scope of variables can be global (all activities can use them) or local (e.g., constrained in a given handler or scope). After partitioning, the result is a set of variables (both local and global) that are accessible by their local orchestrators, and a way to propagate their values to the other orchestrators. To cope with this, we added data-dependence graphs to our approach. As discussed in the literature for liveness analysis, in some configurations it is not possible to detect, at design time, suitable data dependencies between producers and consumers². In such situations, the original WS-BPEL description, with that specific assignment of activities to orchestrators, cannot be partitioned. Thus, we also added an analysis phase, before starting the partitioning process, to understand if the approach is applicable.

Since execution and data flows are similar, it is possible to use the same rules defined for partitioning functional dependencies also for building a distributed data dependence graph. Nevertheless, in a data dependence graph, a node can be reached by several nodes if producers belong to structured activities.

For example, if we considered a `Flow` statement and we suppose that two nodes A and B are executed in parallel and produce the same variable v for node C, in a centralized environment the last activity executed would set the value for v . To preserve such a behavior, we need to add before the consumer node a `Pick Structured Node` with two branches: the first to receive the datum from B and the second from A. The first branch is followed after receiving a message `receiveA`, while the second is triggered by a message `receiveB`. This means that the branch followed is driven by the first node that produces data, but the value is received from the second node.

If we considered `While` statements, producers and consumers can be organized in different ways:

- The producer node is executed before the loop and the consumer is executed within the loop. In this case, we add a `Switch Structured Node` before the consumer. If the iteration is the first, the corresponding switch branch has a node to receive data from the producer; otherwise no operations are performed.
- Producer and consumer are in the loop. The producer is executed before the consumer. This simple case can be directly solved by using the rules already presented in [BMM04].
- The producer is within the loop and the consumer is executed after the end of the loop. In this case, we add a `Basic Node` after the end of the loop to send produced data.
- A producer is before the loop, but another producer is within the loop and after the consumer. To cope with this situation, we add a `Switch Structured Node` before the consumer to distinguish between the first iteration, when the data come from the external producer, and the other iterations, when data come from the internal producer.

Even if our meta-model considers exception handlers as structured activities, they need some specific considerations. First of all, we use the general term exception handler to identify fault, event, and compensation handlers. They comprise two parts: a part that model the raising of the exception, and a part that models the behavior of the handler by using “normal” WS-BPEL activities. Thinking of

²Two WS-BPEL activities are interpreted as producer and consumer if the former sets the value of a given variable that is then used by the latter, without any other activity that changes the value between the two activities.

distributed executions, the second part can be managed with the rules already used for the “normal” behavior, while we need to address more thoroughly the raising of the exception and its propagation to involved orchestrators.

Each exception is associated with a scope and the relevant orchestrators are the same as those involved in the corresponding scope. For this reason, we add a synchronization mechanism at the end of each scope to ensure that if an exception is raised in a scope all orchestrators are properly synchronized. The other operation we need to insert is the propagation of the fault to all involved orchestrators. To cope with this, we propose a solution that is very similar to that used for propagating the value of a variable in `Switch` statements (an orchestrator evaluates it and then propagates the result to the others).

Figure 4 shows how to propagate the fault to all the orchestrators involved in a scope. The `Switch` is necessary to distinguish if the controller is also the catcher of the exception (e.g., a controlled activity failed) or if it is only involved in the scope.

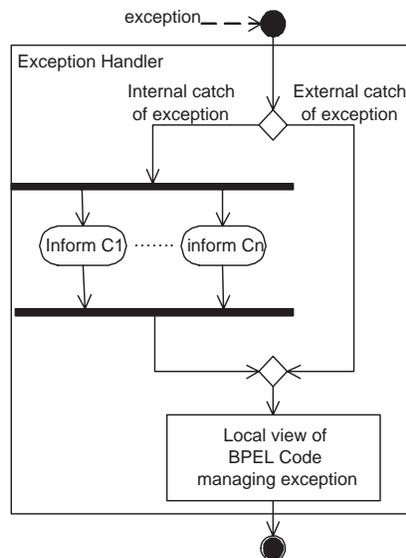


Figure 4: Distributed exception handling

This propagation mechanism, along with the synchronization required before the end of the scope, ensures that each exception is properly forwarded to all interested orchestrators.

To create the local processes, we have a dedicated set of rules. Given the orchestrator for which we want to produce the view, the rules are applied as follows. They: (1) remove all the `Basic Nodes` whose execution is not controlled by the selected orchestrator; (2) translate all the `Structured Nodes`, with the exception of `Sequences`, that do not include “local” activities into `Sequences` with no nodes. The same process, applied iteratively, produces the partial WS-BPEL models for the orchestrators that are part of the system.

3 Example

To explain the approach, we render as workflow a standard procedure defined and realized for carrying out maintenance by the Italian railways. Specifically, we consider the situation in which maintenance and testing of electrical lines and signals have to be performed. This process is partially automatic and involves a central management unit (CMU) and several teams that perform required actions on site. CMU manages the activation/deactivation of railroad traffic on a given track. There are also two teams, one for putting and checking signals and the other one for managing electrical lines. The latter is further divided in the Electric Maintenance team (EMT) and the Electric Test Team (ETT). There is also a control post (CP) that controls the power supply on tracks. In this example, CP only exposes a Web service that allows to inhibit or reactivate electricity on tracks.

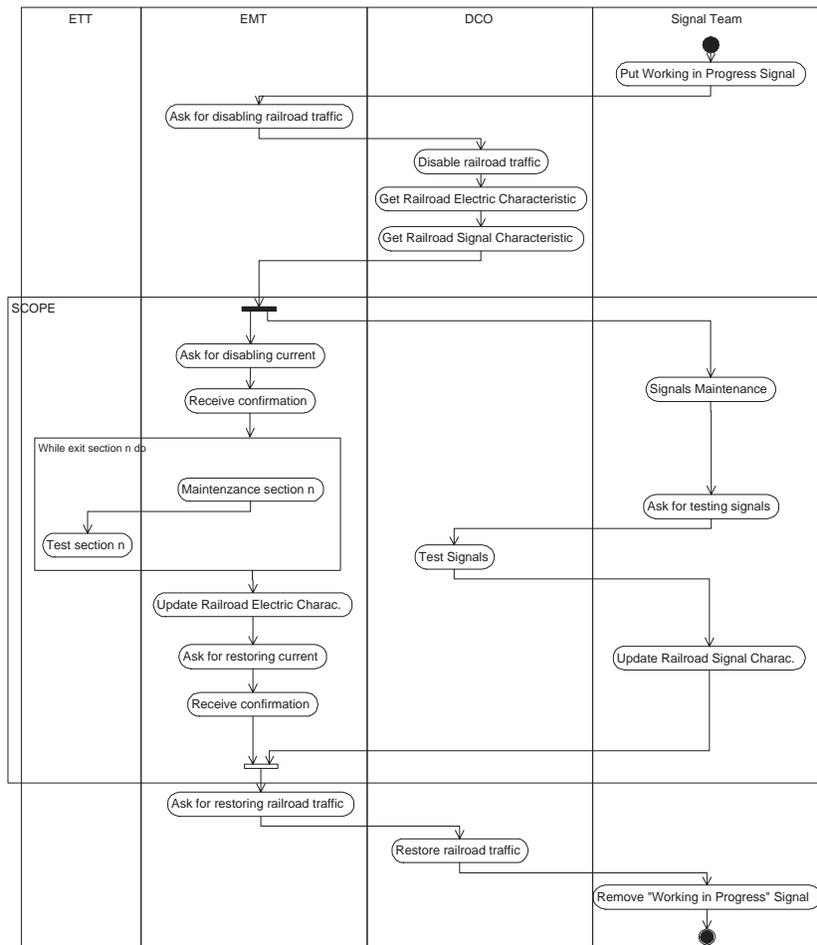


Figure 5: Example workflow

Figure 5 shows the example WS-BPEL process rendered as activity diagram. Each swimlane corre-

sponds to a different controller (orchestrator). For our purposes, we consider each operation as a single task that can be modelled as a WS-BPEL basic activity. A fault handler is associated with the process. It models the possibility of stopping the maintenance activity, and thus resume the availability of the track for the normal train traffic. Notice that the operative teams could be off-line and perform part of their tasks while disconnected.

The workflow starts with the Signal Team that raises a **working in progress** signal on a given track. Then, the EMT asks the CMU for disabling the railroad traffic. CMU disables the traffic and gets from the database the information about the electric characteristics and signals of the considered track. After this, the Electric Team and the Signal Team perform their activities in parallel. Electric maintenance is carried out by asking for the inhibition of the track, by dividing it in sections, and by starting maintenance and testing activities on each section. At the end, the power supply is restored. The maintenance of signals is performed by the Signal Team and then signals are tested by the CMU. At the end of their work, both EMT and Signal Team update the corresponding parts of the database. All operations are enclosed in a scope with an associated fault handler that allows the process to be interrupted to restore the normal traffic on the track. After completing maintenance, the EMT asks for restoring the normal traffic and after the CMU does this, the Signal Team removes the **working in progress** signal.

After explaining the problem, we are ready to apply our partitioning approach. Rules are applied recursively to insert specific coordination activities in the original workflow.

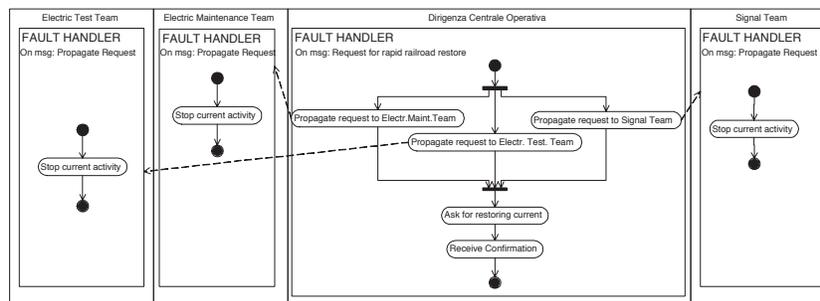


Figure 6: Decentralized fault handler

In our example, we have one fault and it can only be caught by the CMU. With this assumption, Figure 6 shows the fault handlers of the different controllers. After the CMU captures the request for restoring the normal situation, it propagates a message to the other orchestrators to raise an exception in each workflow. After this propagation, it asks the CP for restoring electricity. For security reasons, the procedure would require double confirmation to stop the activities of the operative teams (both CMU and CP would be involved), but this is not important for this example since it would only require that the fault handler be more complex.

The local view of each orchestrator, that is, its sub-process, can then be extracted from the modified workflow: Figure 7 shows the dedicated sub-process for the CMU orchestrator limited to the execution flow.

As far as data are concerned, the example shows that variables railroad electric characteristic and railroad signal characteristic are used by different roles (orchestrators). They store information that

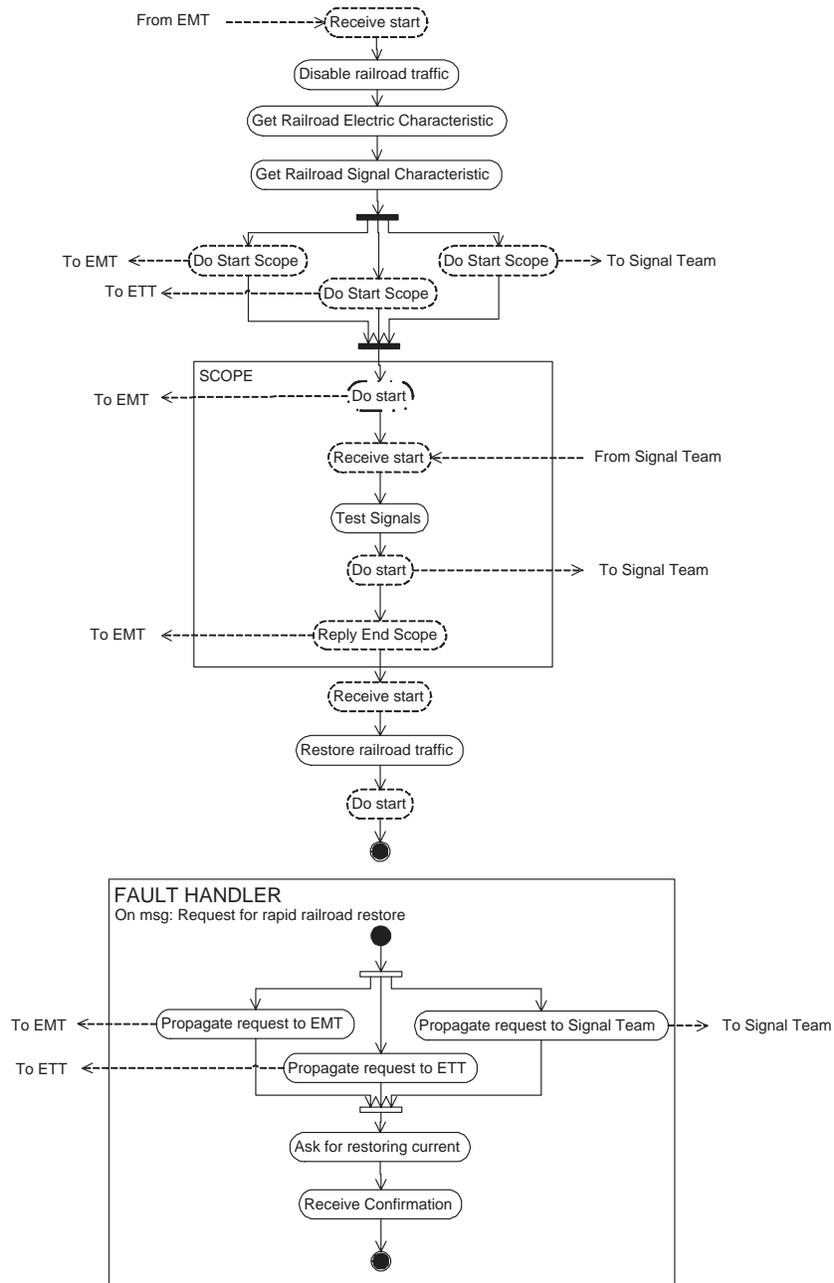


Figure 7: Local view for the CMU

will be updated during the process. In Figure 5, the activities that use these variables are marked with \times and $+$, respectively. In the centralized model, they are global variables, but in the decentralized scenario they have to be propagated to the correct orchestrators. It is possible to determine how these variables are used by building the data flow graph. Then, Figure 8 shows the activities that must be added to the original workflow to manage their propagation.

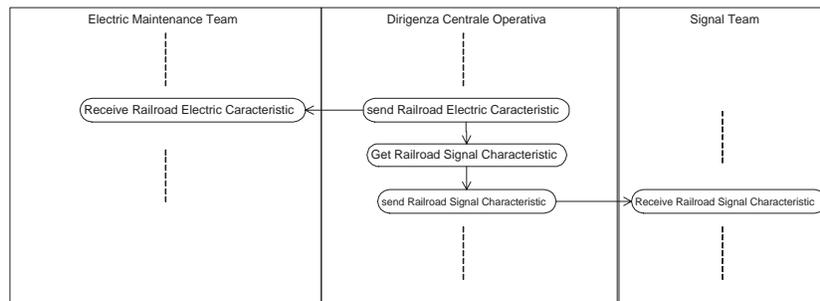


Figure 8: Activities inserted for propagating variables

4 Related work

The problem of workflow partitioning has been well-studied in the field of business process design for some ten years. The issue of workflow distribution still offers interesting aspects to study because of mobile information systems and Web services with the problems that come with them. In [JSH⁺01], the authors present a comparison among the different approaches to distribution.

Cross-Flow [GAHL00] aims at providing high-level support to workflows in dynamically-created virtual organizations. High-level support is obtained by abstracting services and offering advanced cooperation support. Virtual organizations are created dynamically by contract-based match-making between service providers and consumers. In Agent Enhanced Workflow [JOSC98], the agent oriented solution presents the interesting aspect of building execution plans using a goal approach. Event-based Workflow Process Management [EP99] includes an event-based workflow infrastructure and model constructs for addressing time aspects of process management. The main feature of ADEPT [RD98] is the possibility of modifying workflow instances at run-time. MENTOR [MWW⁺98] provides an autonomous workflow engine. In this approach the workflow management system is based on a client-server architecture. The workflow itself is orchestrated by appropriately configured servers, while the applications that invoke workflow activities are executed on the client sites. The METEOR (Managing End to End Operations) [ASC⁺03] system leverages Java, CORBA, and Web technologies to provide support for the development of enterprise applications that require workflow management and application integration. It enables the development of complex workflow applications which involve legacy information systems and that have geographically distributed and heterogeneous hardware and software environments, spanning multiple organizations. It also provides support for dynamic workflow processes, error and exception handling, recovery, and QoS management. Exotica [MAGK95] is characterized by the possibility



of disconnected operations. It does not permit complete decentralization because it maintains a central unit and all operations obey a client/server paradigm. WISE [AFH⁺99] exploits the Web for its engine and offers an embedded fault handler. WAWM [Rie98] focuses on the problems related to the workflow management in wide area networks. Mobile [JB96] is developed to support inter-organizational workflows and is strongly based on modularity. This characteristic alleviates change management and also allows users to customize and extend aspects individually.

The analysis of presented models suggests two different and dual approaches to the problem of workflow coordination. The first approach supports the integration of autonomous and preexisting workflows and it aims mainly at the coordination of different and independent actors. The second approach supports the decomposition of single workflows to support their autonomous execution by means of different engines. Cross-Flow, Agent Enhanced Workflow, Event-based Workflow process Management, Adept, WISE and WAWM belong to the first approach; Mentor, Exotica and Mobile belong to the second one.

Described systems offer three different solutions for the definition of partitioning and allocation rules. The first solution proposes specific definition languages (Cross-Flow, Agent enhanced workflow, Mentor, Exotica). The second approach proposes the extension of workflow languages with distribution rules (Cross-Flow, ADEPT, WISE, WAWM, Mobile). The third approach does not consider the language for distribution rules (Event-based, workflow Process Management). Cross-Flow belongs to more than one class because the distribution rules are split into several definition parts.

Our delegation model supports disconnected components like Exotica, the independence of workflow engines like MENTOR, and the possibility of modifying the workflow instance at run-time like ADEPT. Moreover, we argue that the mobile environment needs a language strongly oriented to the automatic execution like WS-BPEL, but we do not forget the need for lightness that is a mandatory feature if the system runs on portable devices in ad-hoc networks. As far as the definition of rules is concerned, our approach defines partitioning rules, but does not define allocation rules. It demands them to the specific business process and application domain.

Many of these cited approaches do not consider Web services as available instruments for decentralizing business processes. A different comment is for [CCMN04], which presents an approach very similar to ours. The authors use BPEL as workflow model and use the term *Composite Web Service* to refer to a standard workflow. However, they focus on the problem of assigning workflow portions to specific orchestrators to minimize the traffic among nodes, but they do not provide any specific information about the partitioning rules and/or any proof of their validity. They introduce the concepts of *Control Flow Graph* and *Program Dependence Graph* without providing how they refer to WS-BPEL constructs and activities.

5 Conclusions and future work

The paper has presented our approach towards *distributed orchestrations*. It is the continuation of the results already presented in [BMM04], and extends them with the partitioning rules for handling data exchange among orchestrators and for dealing with handlers. Again, the first results are encouraging and are motivating us to continue working on these ideas.

Besides refining the rules, and applying them on further and more complex examples, the key element



of our future work is the definition of the infrastructure to accommodate the execution of our federated WS-BPEL processes. We are working on different solutions that allow us to solve different problems. The first option is the use of standard WS-BPEL engines. This is the simplest solution, but it requires that all synchronization and communication problems among orchestrators be faced and solved in the process specifications. We are thinking of adopting an infrastructure based on a tuplespace middleware (like JavaSpace [Sun04]) to increase the reliability of the supporting infrastructure, cope with the problems that come from producer-consumer analysis, and allow nomadic users to store and retrieve their data in/from dedicated tuples. Finally, we are investigating how to move the synchronization among orchestrators from the WS-BPEL models to the supporting infrastructure by adopting event-based architectures (e.g., WS-Notification) as the means to make the orchestrators communicate.

References

- [AFH⁺99] G. Alonso, U. Fiedler, C. Hagen, A. Lazcano, H. Schuldt, and N. Weiler. WISE: Business to business e-commerce. In *RIDE*, pages 132–139, 1999.
- [Aka04] Akamai Technologies, Computer Associates International, Fujitsu Laboratories of Europe, Globus, Hewlett-Packard, IBM, SAP AG, Sonic Software, TIBCO Software. Web Services Notification. 2004.
- [ASC⁺03] K. Anyanwu, A. Sheth, J. Cardoso, J. Miller, and K. Kochut. Healthcare enterprise process development and integration. *Journal of Research and Practice in Information Technology*, 35(2), 2003.
- [BEA03] BEA and IBM and Microsoft and SAP and Siebel. Business Process Execution Language for Web Services Version 1.1. 2003.
- [Bey92] M. Beyer. *AGG1.0 - Tutorial*. Technical University of Berlin, Department of Computer Science, 1992.
- [BH02] L. Baresi and R. Heckel. Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In *Proceedings of the First International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 402–429. Springer-Verlag, 2002.
- [BMM04] L. Baresi, A. Maurino, and S. Modafferi. Workflow partitioning in mobile information systems. In Kluwer, editor, *In Proc. of IFIP TC8 Working Conference on Mobile Information Systems*, volume 158 of *IFIP International Federation for Information Processing*, 2004.
- [BMM05] Luciano Baresi, Andrea Maurino, and Stefano Modafferi. Partitioning rules for ws-bpel processes. Technical report, Politecnico di Milano, 2005.
- [CCMN04] G.B. Chafle, S. Chandra, V. Mann, and M.G. Nanda. Decentralized orchestration of composite web services. In *In Proc. of the Int. World Wide Web conference on Alternate track papers & posters*, pages 134–143, New York, NY, USA, 2004. ACM Press.



- [CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) 1.1*. W3C, March 2001.
- [EP99] J. Eder and E. Panagos. Towards distributed workflow process management. In *In proc. of Workshop on cross-Organizational Workflow Management and Coordination*, San Francisco, USA, 1999.
- [Ga03] T. Gardner and al. Draft uml 1.4 profile for automated business processes with a mapping to the bpel 1.0. IBM alphaWorks, 2003.
- [GAHL00] P. Grefen, K. Aberer, Y. Hoffner, and H. Ludwig. Crossflow: Cross-organizational workflow management in dynamic virtual enterprises. *International Journal of Computer Systems Science & Engineering*, 15(5):277–290, 2000.
- [JB96] S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation*. International Thomson, 1996.
- [JOSC98] D. Judge, B. Odgers, J. Shepherdson, and Z. Cui. Agent enhanced workflow. *BT Technical Journal*, (16), 1998.
- [JSH⁺01] S. Jablonski, R. Schamburger, C. Hahn, S. Horn, R. Lay, J. Neeb, and M. Schlundt. A comprehensive investigation of distribution in the context of workflow management. In *In proc. of International Conference on Parallel and Distributed Systems ICPADS*, Kyongju City, Korea, 2001.
- [MAGK95] C. Mohan, G. Alonso, R. Gunthor, and M. Kamath. Exotica: A research perspective of workflow management systems. *Data Engineering Bulletin*, 18(1):19–26, 1995.
- [MWW⁺98] P. Muth, D. Wodtke, J. Weisenfels, A. Kotz Dittrich, and G. Weikum. From centralized workflow specification to distributed workflow execution. *Journal of Intelligent Information Systems*, 10(2):159–184, 1998.
- [Nic04] Nickolaos Kavantzias and David Burdett and Greg Ritzinger. *Web Services Choreography Description Language Version 1.0*. 2004.
- [Obj02] Object Management Group. *Meta Object Facility (MOF) Specification - v.1.4*. Technical report, OMG, March 2002.
- [RD98] M. Reichert and P. Dadam. Adeptflex – supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.
- [Rie98] G. Riempp. *Wide Area Workflow Management*. Springer, London, UK, 1998.
- [Sun04] Sun Microsystems. *Jini network technology*, 2004. www.sun.com/software/jini/.
-



EMF Model Refactoring based on Graph Transformation Concepts

Enrico Biermann *, Karsten Ehrig **, Christian Köhler *, Günter Kuhns *, Gabriele Taentzer *,
Eduard Weiss *

*Department of Computer Science, Technical University of Berlin,
Germany, {enrico,bunjip,jaspo,gabi,eduardw}@cs.tu-berlin.de

**Department of Computer Science, University of Leicester, UK, karsten@mcs.le.ac.uk

***Abstract.** The Eclipse Modeling Framework (EMF) provides a modeling and code generation framework for Eclipse applications based on structured data models. Within model driven software development based on EMF, refactoring of EMF models become a key activity. In this paper, we present an approach to define EMF model refactoring methods as transformation rules being applied in place on EMF models. Performing an EMF model refactoring, EMF transformation rules are applied and can be translated to corresponding graph transformation rules, as in the graph transformation environment AGG. If the resulting EMF model is consistent, the corresponding result graph is equivalent and can be used for validating EMF model refactoring. Results on conflicts and dependencies of refactorings for example, can help the developer to decide which refactoring is most suitable for a given model and why.*

Keywords: Model refactoring, Eclipse Modeling Framework, graph transformation

1 Introduction

In the world of model-driven software development, the Eclipse Modeling Framework (EMF) [EMF06] is becoming a key reference. It is a framework for describing class models and generating Java code; it supports the creation, modification, storage, and loading of model instances. Moreover, it provides generators to support the editing of EMF models.

EMF unifies three important technologies: Java, XML, and UML. Regardless of which one is used to define a model, an EMF model can be considered as the common representation that subsumes the others. That means defining a transformation approach for EMF, it will become also applicable to the other technologies.

Refactoring within the model-driven software development process means to refactor the corresponding

models. Basing the model-driven approach on EMF models, refactoring of EMF models becomes a key activity which should be supported. Since EMF unifies three different technologies, i.e. Java, XML and UML, the EMF refactoring can also be used to restructure Java programs, XML schemas and UML models. Considering especially UML, a number of refactoring methods are already available, see e.g. [SPTJ01], [Por03], [MB05], which all cover at least refactoring of class models. Since EMF models resemble very much UML class models, UML refactoring methods can also be considered for EMF model refactorings.

Different approaches have been considered for model refactorings which can be categorized as model transformations in general, optimizing models of a given modeling language. Most of the refactoring approaches presented (e.g. [SPTJ01], [MB05]) use OCL constraints to describe the pre- and post-conditions of refactorings in a declarative way. Another kind of approaches use transformation rules (e.g. [Por03]). In [MT04], declarative approaches based on pre-/post-conditions are compared with graph transformation approaches.

In this paper, we consider two selected refactorings of instances of the Ecore model being the EMF meta model and as such also an EMF model. Besides the Ecore model, we could also choose any other EMF model, such as the UML2 model, for refactoring. Our transformation approach is based on graph transformation and adapted to EMF models.

In our running example, we consider an EMF model which stores the abstract syntax of simple place/transition Petri nets and refactor it in order to get a more object-oriented model. (See the original Petri net model in Fig. 1.) This model is restructured by pulling up the common attribute "name" of classes "Place", "Transition" and "PetriNet" to a new superclass "NamedElement". (See the refactored model in Fig. 2.) In the following, we consider "create superclass" and "pull up attribute" as sample EMF model refactorings.

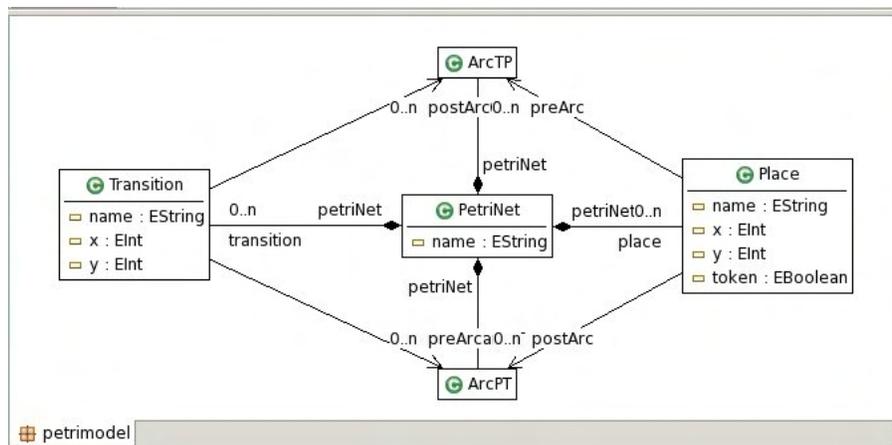


Figure 1: Petri net model before refactoring

EMF model refactoring can be considered as endogenous model transformation [MVG06] performing some kind of model optimization. When applying a refactoring method to an EMF model, this model shall be modified, i.e. it shall be transformed in-place. Considering the current transformation approaches

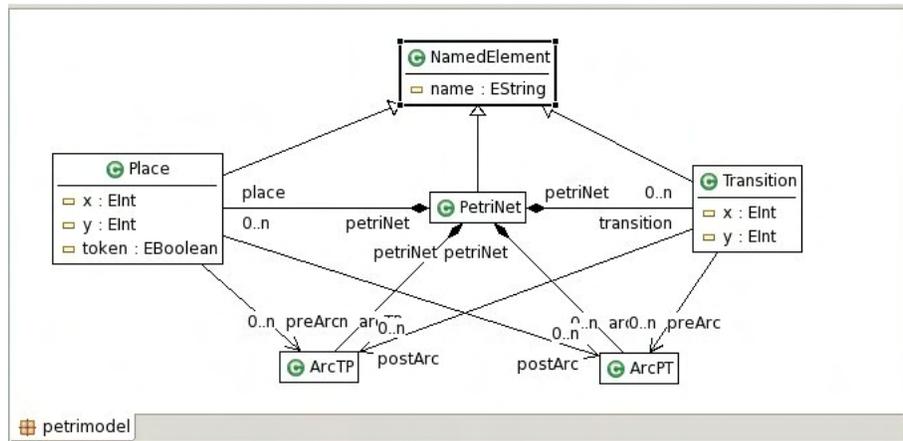


Figure 2: Petri net model after refactoring

for EMF such as Tefkat [LS05], ATL [JK06], MTF [MTF05], Merlin [Mer06], and the transformation engines developed within the Eclipse Project General Model Transformer (GMT), a transformation engine for in-place transformation and with validation facilities for model transformations is not yet available.

Therefore, we recently developed an endogenous EMF model transformation engine [BEK⁺] which can perform in-place model transformations, based on graph transformation concepts. The transformation description can be compiled to Java code using those EMF classes already generated. Furthermore, it is possible to translate the rules to AGG [AGG06], a tool environment for algebraic graph transformation where the transformation might be further analyzed.

The analysis of transformations and transformation rules is helpful in deciding what to refactor and when. Termination of refactoring operations as well as conflicts and dependencies between different refactorings are important issues to be analysed to inform the user about the refactorings which can be performed. The dependency analysis has already been considered in [MTR04].

2 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [EMF06] provides a modeling and code generation framework for Eclipse applications based on structured data models. The modeling approach is similar to that of MOF, actually EMF supports Essential MOF (EMOF) as part of the OMG MOF 2.0 specification [EMO06]. The type information of sets of instance models is defined in a so-called core model corresponding to metamodel in EMOF. The core or metamodel for core models is the EcCore model. It contains the model elements which are available for EMF core models in principle. In Fig. 3, the main part of EcCore (without attributes) is shown. The kernel model contains elements EClass, EDataType, EAttribute and EReference. These model elements are needed to define classes by EClass, their attributes by EAttribute and interrelations by EReference. EClasses can be grouped to EPackages which might be again structured into subpackages. In addition, each model element can be annotated by EAnnotation. Fur-



3.1 EMF Model Transformation Approach

Basically, an EMF transformation is a rule-based modification of an EMF source model resulting in an EMF target model. Both, the EMF source and target models are typed over an EMF core model which itself is again typed over Ecore. Refactoring can take place on two levels: (1) Refactoring rules are typed over the Ecore model and are applied to EMF models. The running example of this paper containing refactorings of a Petri net model (being an EMF core model) is of this kind. (2) Refactoring rules are typed over some EMF core model (e.g. the Petri net model in Fig. 1) and refactor EMF instance models (e.g. Petri net instance models).

A *Transformation System* consists of a set of transformation rules. Furthermore, it has a link to the core model its instances are typed over. Rules are expressed mainly by two object structures LHS and RHS, the left and right-hand sides of the rule. Furthermore, a rule has mappings between objects of the LHS and the RHS indicated by numbers preceding the class names. The left-hand side LHS represents the pre-conditions of the rule, while the right-hand side RHS describes the post-conditions. Those objects of the LHS which are mapped to the RHS, describe a structure part which has to occur in the EMF source model, but which is not changed during the transformation. All objects of the LHS not mapped to the RHS define the part which shall be deleted, and all symbols and links of the RHS to which nothing is mapped, define the part to be created.

The applicability of a rule can be further restricted by additional application conditions. As already mentioned above, the LHS of a rule formulates some kind of positive condition. In certain cases also *negative application conditions* (NACs) which are pre-conditions prohibiting certain object structures, are needed. If several NACs are formulated for one rule, each of them has to be fulfilled. A NAC is again an object structure which is the target of a mapping from the LHS. This feature is useful to prohibit structures connected to the LHS.

The rule's LHS or a NAC may contain constants or variables as attribute values, but no Java expressions, in contrast to an RHS. A NAC may use the variables already used in the LHS or new variables declared as input parameters. The scope of a variable is its rule, i.e. each variable is globally known in its rule. The Java expressions occurring in the RHS, may contain any variable used within the LHS or declared as input parameter. Multiple usage of the same variable is allowed and can be used to require equality of values.

A rule-based transformation system may show two kinds of non-determinism: (1) for each rule several matches may exist, and (2) several rules may be applicable. There are techniques to restrict both kinds of choices. The choice of matches can be restricted by using input parameters. Moreover, some kind of control flow on rules can be defined by applying them in a certain order. For this purpose, rules are equipped with layers. All rules of one layer are applied as long as possible (in any order) before going over to the next layer. The transformation stops after having executed the last layer.

To apply the defined transformation rules on a given EMF model, we either select and apply the rules step-by-step, or take the whole rule set and let it apply as long as possible. A transformation step with a selected rule is defined by first finding a match of the LHS in the current instance model. A pattern is matched to a model if its structure can be found in the model such that the types and attribute values are compatible. In general, a pattern can match to different parts of a model. In this case, one of the possible matches has to be selected, either randomly or by the user.

Performing a transformation step which applies a rule at a selected match, the resulting object structure is constructed in two passes: (1) all objects and links present in the LHS but not in the RHS are deleted; (2) all object and links in the RHS but not in the LHS are created. A transformation, more precisely a transformation sequence, consists of zero or more transformation steps.

3.2 Selected Refactoring Methods for EMF Models

Based on the presented transformation approach for EMF models we show two selected refactoring methods for EMF models. All transformation rules are typed over the Ecore model, in more detail over the Ecore section shown in Fig. 3. In the following, we define the simple refactorings “create superclass” and “connect superclass” where a new superclass is created for a given class and can become superclass of further classes. Moreover, the complex refactoring “pull up attribute” is shown. If each subclass contains an attribute with the same name and type, it can be pulled up to their common superclass.

Refactoring rule “CreateSuperclass(EString c, EString s, boolean a)” in Fig. 4 has parameters “c” and “s” to determine the name of the child class and of the new super class. Moreover, we have to decide if the new super class shall be abstract. The LHS describes the pattern to be found for refactoring consisting of a class which will be the child and the package it belongs to. The RHS shows the new pattern after refactoring where a new class with name “s” has been created which is the super class of the given class. The super class shall be contained in the same package as the class its subclass. Fig. 4 shows the left and the right-hand sides of the rule. Objects which are preserved occur in both parts. If two objects correspond to each other, they are colored and numbered equally.

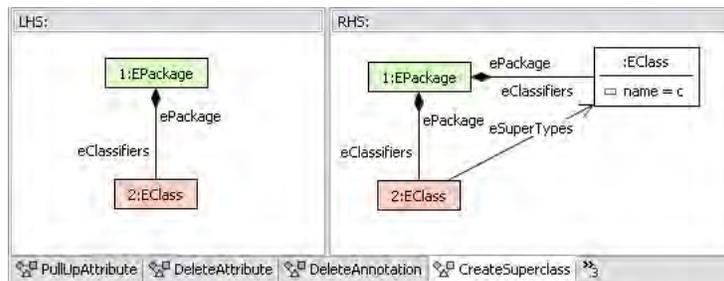


Figure 4: Rule “CreateSuperclass”

After creating a super class, it might become super class of more than one class which can be restructured by rule “ConnectSuperclass(EString s, EString c)” in Fig. 5. A class is allowed to become subclass of a given class, if that class does not have attributes and references yet. These additional conditions are expressed by two NACs “No Attribute” and “No Reference” which check that the class does not have an EAttribute and does not have an EReference to some class.

Refactoring “PullUpAttribute” is more complex, i.e. it cannot be defined by just one rule, but four rules are needed to check the complex pre-condition, to do the kernel refactoring, and to make the model consistent afterwards. For checking the pre-condition, rule “CheckAttribute(EString c, EString a)” in Fig. 6 checks for the class named “c” if there is a subclass not containing an attribute named “a”. This

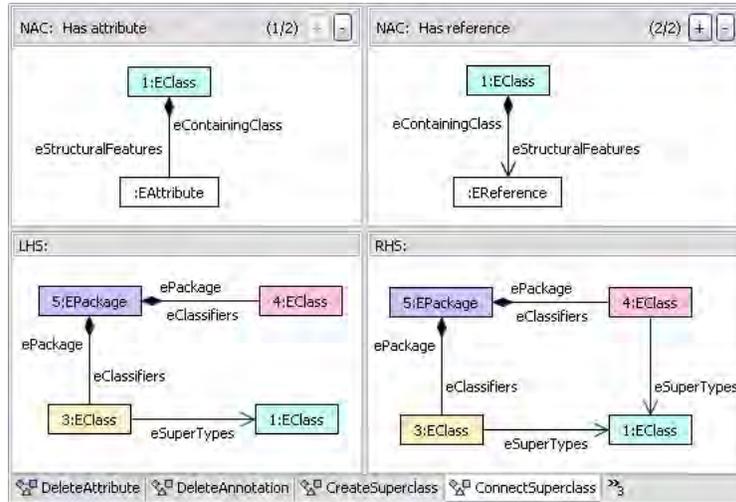


Figure 5: Rule "ConnectSuperclass"

rule can be applied at most once, since there are NACs which check if there is already a subclass with this annotation. Thereafter, we try to apply rule "PullUpAttribute(EString c, EString a)" in Fig. 7. If there is no subclass of the class named "c" which has an annotation with source "no attribute" and if the class named "c" has not already an attribute named "a", it looks for a subclass which has an attribute named "a". After the refactoring, an existing attribute with name "a" is pulled up. This rule is applicable at most once. Thereafter, NAC "Attribute already pulled up" will not be satisfied anymore. NAC "Attribute not in all sub-types" checks a necessary pre-condition.

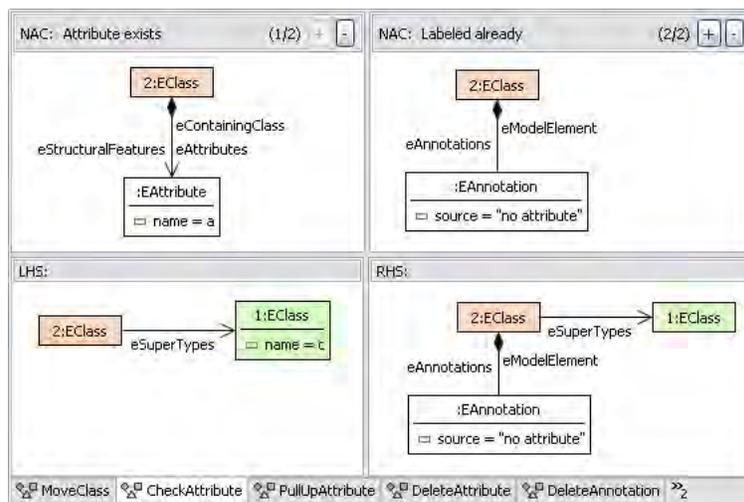


Figure 6: Rule "CheckAttribute"

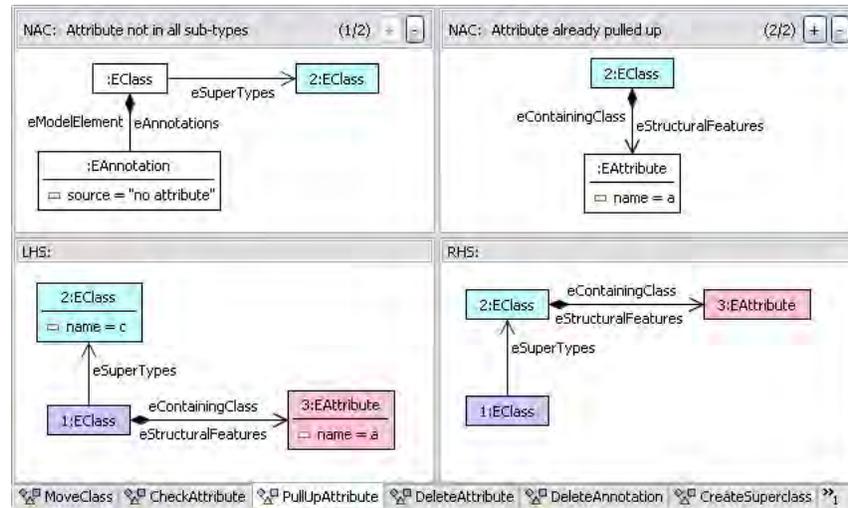


Figure 7: Rule "PullUpAttribute"

If "PullUpAttribute" was successful, i.e. there is no subclass with a corresponding annotation, all attributes named "a" being still contained in subclasses have to be deleted. This is done by rule "DeleteAttribute(EString c, EString a)" in Fig. 8 applying it as long as possible. Finally, if the refactoring was not successful, all new annotations have to be deleted again which is performed by rule "DeleteAnnotation()" in Fig. 9. The application control for these rules just described can be realised by putting each of the rules to consecutive layers in the order of description. (See Table 1.)

Layer	Rule
1	CreateSuperclass, ConnectSuperclass, CheckAttribute
2	PullUpAttribute
3	DeleteAttribute
4	DeleteAnnotation

Table 1: Control flow for refactoring "pull up attribute"

Compared to the first refactorings, the implementation of refactoring "pull up attribute" looks rather complicated. Four rules instead of one are needed, since our approach does not support complex pre-conditions which allow to check for-all-conditions. For example, refactoring "pull up attribute" is allowed only, if all direct subclasses contain that attribute to be pulled up. After this refactoring, the resulting model has to be updated in the sense that the subclasses do not have to contain that attribute anymore. Thus, rules should have a for-all-operator for deletion and/or creation of graph parts. We restricted our transformation approach such that for-all-conditions and -operators are supported, because we provide algebraic graph transformation as formal basis for validation purposes, provided by AGG. Since there are also formal concepts for graph transformation with for-all-conditions and -operators around, it is up to



is changed, the corresponding code and documentation has to be changed accordingly (see [MT04] for further information). Following the model-driven software development paradigm thoroughly, code would be changed accordingly as soon as the code generator uses the refactored model. Thus, consistency between model and code could be easily achieved in this case. (2) More basically, consistency of model refactorings should also mean a kind of syntactic correctness in the sense that the refactored model is still an element of the given modeling language and fulfills certain validation properties. In the following, we concentrate on this second kind of consistency, while the first kind is out of scope of this paper.

Similarly to MOF, modeling languages are defined with EMF by a class model defining the model elements and their relations. Since all refactoring rules are typed over the EMF core model, i.e. the meta model, the refactored model is still correct wrt. its meta model. If the meta model contains additional consistency constraints, they have to be checked after each refactoring to make sure that the resulting model is still consistent.

4.1 Consistency with Graph Transformation

To open up the possibility for formal validation of EMF model refactorings, another kind of consistency is needed. In the following, we consider EMF model refactorings as consistent if they can be compared with graph transformations. In this case, the formal analysis techniques for graph transformation become available also for EMF refactoring. Results on conflicts and dependencies of refactorings for example, can help the developer to decide which refactoring is most suitable for a given model and why.

Although EMF models show a graph-like structure and can be transformed similarly to graphs [EEPT06], there is a main difference in between. In contrast to graphs, EMF models have a distinguished tree structure which is defined by the containment relation between their classes. An EMF model should be defined such that all its classes are transitively contained in the root classes. Since an EMF model may have non-containment references in addition, the following question arises: What if a class which is transitively contained in a root class, has non-containment references to other classes not transitively contained in some root class? In this case we consider the EMF model to be inconsistent.

A transformation can make an EMF model inconsistent, if its rule deletes one or more objects. For example an inconsistent situation occurs, if one of these objects transitively contains an object referred to by a non-containment link. To restore the consistency, all objects to be deleted have to be determined. Thereafter, all non-containment references to these indicated objects have to be removed, too. To ensure consistent transformations only, rules which delete objects or containment links or redirect them, have to be equipped with additional NACs.

If a containment link is deleted, the corresponding contained object has to be deleted, too. This object is not allowed to contain further objects. This constraint has to be required with a separate NAC. If a containment link is set or redirected and a maximum multiplicity is set on the container's end, an application condition is needed which checks that this multiplicity has not yet been reached, i.e. there do not exist objects which would be without container after rule application.

Similarly to the handling of deleted structures, consistency recovery is also applied to newly created objects. If a rule creates objects which are not transitively contained in one of the root objects, the consistency recovery will remove these objects at the end of a rule application. It is easily possible to forbid the application of those rules entirely, since inconsistencies on creation of objects can be determined

statically.

Our visual editor for EMF transformation rules is able to check the consistency with graph transformation discussed above. Every time an inconsistency is found, the inconsistent part is highlighted and the inconsistency is further described to inform the user.

4.2 Consistency of Selected EMF Model Refactorings

All selected refactoring rules are typed over the Ecore model, thus also the refactored EMF models will be typed over Ecore.

Considering the consistency of the selected EMF model refactorings, we define all objects of type “EPackage” as root objects. Now we check the consistency of those refactoring rules defined in the previous section: Rules “CreateSuperclass”, “ConnectToSuperclass”, “CheckAttribute”, and “PullUpAttribute” are consistent by definition. Rule “DeleteAttribute” could cause an inconsistent model in principle, but does not do so, since “EAttribute” objects cannot have children and all possible links occur in the LHS. In contrast, rule “DeleteAnnotation” defines the deletion of an object which may have children due to the Ecore model, but does not have any applying only the refactoring rules. To ensure a consistent rule application only, rule “DeleteAnnotation” could be extended by a NAC checking that there is no EObject connected to the given EAnnotation. In this way we get a set of consistent refactoring rules only which can be translated to a set of corresponding graph transformation rules for further validation. In this case, the EMF refactoring rules can be translated to AGG, a tool environment for algebraic graph transformation where they might be further analyzed. Since all refactoring rules in the running example preserve the consistency of EMF models, analysis techniques such as critical pair analysis, termination checks, etc. are available also for EMF model refactorings. For example in [MTR04], critical pair analysis was used to detect conflicts and dependencies between refactorings of class models.

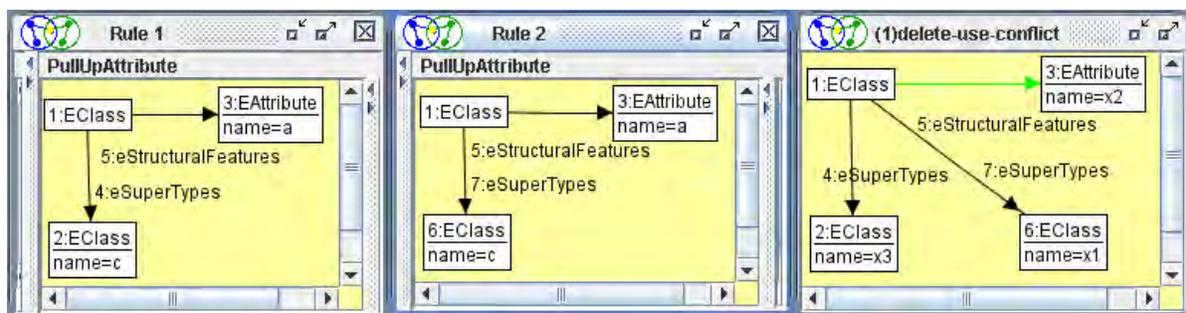


Figure 10: Sample critical pair in AGG

As a sample critical pair a “delete-use-conflict” is shown in Fig. 10 produced by the critical pair analysis of AGG between rule “PullUpAttribute” two times. The overlapping graph at the right-hand side of Fig. 10 depicts a situation where edge “5:eStructuralFeatures” is critical, since it is deleted during the first application and cannot be used again during the second application of “PullUpAttribute”. Considering our example, this situation is not critical, since one and the same “EAttribute” is intended to be pulled up only once.



5 Conclusion

In this paper we presented an approach to specify the refactoring of EMF models by endogenous, in-place EMF model transformation. We use a recently developed EMF model transformation engine [BEK⁺] which is based on algebraic graph transformation concepts. Due to this formal basis, a formal analysis of conflicts and dependencies of EMF model refactorings can be performed. A necessary presumption for this kind of formal validation is the consistency of EMF refactorings with graph transformations. We could show that the selected EMF model refactoring rules fulfill this kind of consistency.

It is up to future work to investigate further EMF model refactorings, to answer the following questions: How far are EMF refactorings similar to UML refactorings? Which of the EMF refactorings are consistent with graph transformation (as defined above) and thus, can be formally analysed concerning conflicts and dependencies?

Future work also includes the development of a comprehensive environment for EMF model refactoring. A visual editor and a transformation engine with code generator presented in [BEK⁺] are already available at <http://tfs.cs.tu-berlin.de/emftrans>. Further tools such as a visual debugger and validation tools would be helpful.

References

- [AGG06] *AGG-System* <http://tfs.cs.tu-berlin.de/agg/>, 2006.
- [BEK⁺] E. Biermann, K. Ehrig, G. Kuhns, C. Köhler, G. Taentzer, and E. Weiss. Graphical Definition of Rule-Based Transformation in the Eclipse Modeling Framework. In *Springer LNCS. 9th Int. Conf. on Model Driven Engineering Languages and Systems*. To appear.
- [EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer, 2006.
- [EMF06] *Eclipse Modeling Framework (EMF)* <http://www.eclipse.org/emf>, 2006.
- [EMO06] *Essential MOF (EMOF) as part of the OMG MOF 2.0 specification* <http://www.omg.org/docs/formal/06-01-01.pdf>, 2006.
- [JK06] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers, LNCS 3844*, edited by Jean-Michel Bruel. Springer Berlin / Heidelberg, pages 128–138, 2006.
- [LS05] M. Lawley and J. Steel. Practical Declarative Model Transformation With Tefkat. In J. Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*. Springer, LNCS 3844, 2005.
- [MB05] Slavisa Markovic and Thomas Baar. Refactoring ocl annotated uml class diagrams. In *MoDELS*, pages 280–294, 2005. URL: <http://igl.epfl.ch/pub/Papers/baar-2005-models.pdf>.



- [Mer06] *Merlin Generator* <http://sourceforge.net/projects/merlingenerator/>, 2006.
- [MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.
- [MTF05] *IBM Model Transformation Framework* <http://www.alphaworks.ibm.com/tech/mtf>, 2005.
- [MTR04] T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts using Critical Pair Analysis. In *In R. Heckel and T. Mens, editors, Proc. Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra'04), Satellite Event of ICGT'04, Rome, Italy, 2004.*
- [MVG06] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. In *Proc. International Workshop on Graph and Model Transformation (GraMoT'05)*, number 152 in *Electronic Notes in Theoretical Computer Science*, Tallinn, Estonia, 2006. Elsevier Science.
- [Por03] Ivan Porres. Model Refactorings as Rule-Based Update Transformations. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 159–174. Springer, 2003. URL: <http://citeseer.ist.psu.edu/porres03model.html>.
- [SPTJ01] Gerson Sunye, Damien Pollet, Yves Le Traon, and Jean-Marc Jezequel. Refactoring UML models. In *The Unified Modeling Language*, pages 134–148, 2001.



Exogenous Model Merging by means of Model Management Operators

Artur Boronat *, José Á. Carsí *and Isidro Ramos *

*Technical University of Valencia

Information Systems and Computation Department

Camí de Vera, s/n. 46022, València (Spain)

***Abstract.** In Model-Driven Engineering, model merging plays a relevant role in the maintenance and evolution of model-based software. Depending on the amount of metamodels involved in a model merging process, we can classify model merging techniques in two categories: endogenous merging, when all the models to be merged conform to the same metamodel; and exogenous merging, when the models to be merged conform to different metamodels. MOMENT (MOfdel manageMENT) is a framework that is integrated in the Eclipse platform, and provides a collection of generic set-oriented operators to manipulate MOF models, following the Model Management discipline. In this paper, we study how model transformations are useful in a model merging process and we provide a solution for both kinds of model merging by means of model management operators and the QVT Relations language.*

Keywords: Model-Driven Architecture, Model Management, Exogenous Model Merging, QVT Relations

1 Introduction

Software merging is an essential aspect of the maintenance and evolution of large-scale information systems. Information systems can be specified by means of models in Model-Driven Engineering. Models collect the information that describes the information system at a high level of abstraction, which permits the development of the application in an automated way using generative programming techniques. The consolidation of the Meta-Object Facility standard [OMG04] as a four-layer architecture, where metamodels can be specified as a set of syntactical well-formedness rules to define models, permits the definition of modeling domains where merging processes can be performed. A model merging process can be defined over a metamodel. Then, any two well-formed models in this metamodel can be merged. Traditionally, the tasks that are involved in this process have usually been solved in an ad-hoc manner for a specific context or metamodel: relational databases [BLN86, BDK92], XML schemas [Beh00], OWL-DL ontologies [HM05], aspect-oriented modeling [SGS⁺04], UML models [OWK03], etc.

[Men02] presents a classification of merge approaches, where domain independence and customizabil-



ity of a generic merge operator to a specific domain are desired features. However, the definition of metamodels by means of a common metamodeling language (like MOF, or any MOF-like implementation) is a desired feature that should be preserved on the grounds that it permits the development of generic infrastructures to manipulate models.

Following this direction, Model Management [BHP00] is a new emergent discipline that pursues an abstract reusable solution for problems of this kind, independently of the metamodel under study. The Model Management discipline deals with software artifacts by means of generic operators that do not depend on their internal implementation because they work on mappings between models [Ber03]. These operators treat models as first-class citizens and increase the level of abstraction of the solution avoiding programming tasks and improving the reusability of the solution.

As stated in [BLN86], a model merging process consists of three main phases: a model comparison phase, where elements of different models that are equivalent are found; a consistency checking phase, where conflicts that may appear if we merge equivalent elements are identified, defining a conflict resolution strategy to eliminate them; and a merging phase, where the equivalent elements that are found in the first step are merged taking into account the conflict strategy defined in the second step.

Generic model merging approaches provide support for these three phases in different ways. [AP03] uses MOF identifiers to compare elements in different versions of a same base model. [BP03, BCE⁺06] provide a set of model management operators to define equivalence relationships between elements of different models by means mappings, which are used by a merge operator later on. [KPP06] proposes several domain-specific languages to define model comparison and model merging over metamodels. The model comparison language permits the definition of equivalence relationships between elements of a metamodel that can be applied over elements of the corresponding models afterwards. The model merging language embeds the comparison language so that these equivalence relationships can be used in the merging process.

In our approach, we propose a set of model management operators that use the QVT Relations language [OMG05] to perform model comparison and model transformation. In a model merging process where two models are involved, the comparison phase is achieved by defining relations between elements of the same metamodel. The consistency phase is solved by defining a model transformation that takes the two models to be merged as input models. Finally, the merging phase is performed by a generic operator that uses the QVT Relations programs defined in the previous phases. Thus, we enhance the use of the QVT Relations language within the Model Management field, avoiding the definition of a new DSL for every model management operator. In this paper, we show how this approach can be used by providing an example of exogenous model merging, where the models to be merged conform¹ to different metamodels.

The structure of the paper is as follows: Section 2 presents the exogenous model merging problem; Section 3 introduces the *ModelGen* operator for model transformation; Section 4 introduces the *Merge* operator for model merging; Section 5 provides the solution for the example in Section 2; Section 6 provides some related works. Finally, Section 7 summarizes the main contributions of the paper.

¹A model conforms to a metamodel if it is syntactically well-formed by using the constructs of the metamodel.

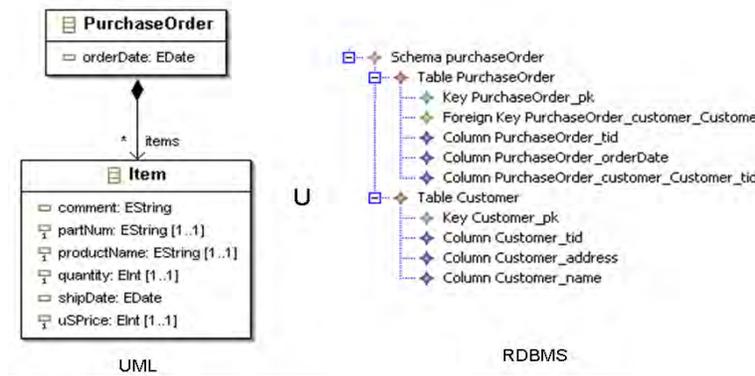


Figure 1: Exogenous model merging of a UML model and a relational schema.

2 Exogenous Model Merging Scenario

When two models are merged, an equivalence relation must be defined between their corresponding metamodels, associating their elements using a set of relationships. These relationships are used to identify equivalent elements in different models in order to avoid duplicated information in the merged model.

Generic approaches to merge models use this concept of equivalence relation, but they do not usually differentiate between an endogenous and an exogenous model merging. In Fig. 1, we provide an example of exogenous model merging: the integration of a UML model and a relational schema. We have used the Ecore metamodel [BBM03] as an implementation for the UML class diagram metamodel, and the relational metamodel that appears in the Query/View/Transformation (QVT) standard specification [OMG05]. In Fig. 1, the relational schema is shown in a tree-like form.

In this paper, we use this example to show that an exogenous model merging process is a generalization of an endogenous model merging process. Therefore, it can be broken down into simpler processes, which can be solved by means of model management operators. Our approach for solving the example consists of two steps: a model transformation that permits representing the UML model as a relational schema; and a model merging between relational schemas. We present how we deal with model transformation and endogenous model merging in the following sections.

3 The QVT Relations Language and the *ModelGen* Operator

In the QVT Relations language, a model transformation is defined among several metamodels, which are called the domains of the transformation. A QVT transformation is constituted by QVT relations, which become declarative transformation rules. A QVT relation specifies a relationship that must hold between the model elements of different candidate models. The direction of the transformation is defined when it is invoked by choosing a specific domain as target. If the target domain is defined in the QVT



transformation as *enforce*, a transformation is performed by creating the corresponding elements in the target model. If the target domain is defined as *checkonly*, just a checking is performed without creating any new element in the target model. Both kinds of transformations are used in our approach.

A relation can be also constrained by two sets of predicates, a *when* clause and a *where* clause. The *when* clause specifies the conditions under which the relationship needs to hold. The *where* clause specifies the condition that must be satisfied by all model elements participating in the relation.

A transformation contains two kinds of relations: top-level (marked with the *top* keyword) and non-top-level. The execution of a transformation requires that all its top-level relations hold, whereas non-top-level relations are required to hold only when they are invoked directly or transitively from the *where* clause of another relation.

As example, we have taken the *UmlToRdbms* transformation that is presented in the MOF QVT final specification². The top relation below specifies the transformation of a *Class* into a *Table*. By means of the *where* clause, the relation *ClassToTable* needs to hold only when the *PackageToSchema* relation holds between the package containing the class and the schema containing the table. By means of the *when* clause, the *ClassToTable* relation holds, the relation *AttributeToColumn* must also hold.

```

top relation ClassToTable {
  className: String;
  checkonly domain.ecoreDomain c: EClass {
    ePackage = p:EPackage {},
    name=className
  };
  enforce domain.rdbmsDomain t: Table {
    schema = s:Schema {},
    name = className,
    column = cl:Column {
      name = className + '_tid',
      type = 'NUMBER'
    },
    key = k:Key {
      name = className + '_pk',
      column=cl
    }
  };
  when {
    PackageToSchema(p, s);
  }
  where {
    AttributeToColumn(c, t, className);
  }
}

```

In MOMENT, a model transformation can be applied to several source models, which may or may not conform to the same metamodel. When the transformation is invoked, it generates one target model and a set of traceability models. A traceability model contains a set of traces that relate the elements of the source model to the elements of the target model, indicating which transformation rule has been

²In this paper, we are using a version of this transformation in which we consider Ecore as an implementation of the UML Class Diagram metamodel. The version of the transformation that is used is presented in Appendix B.

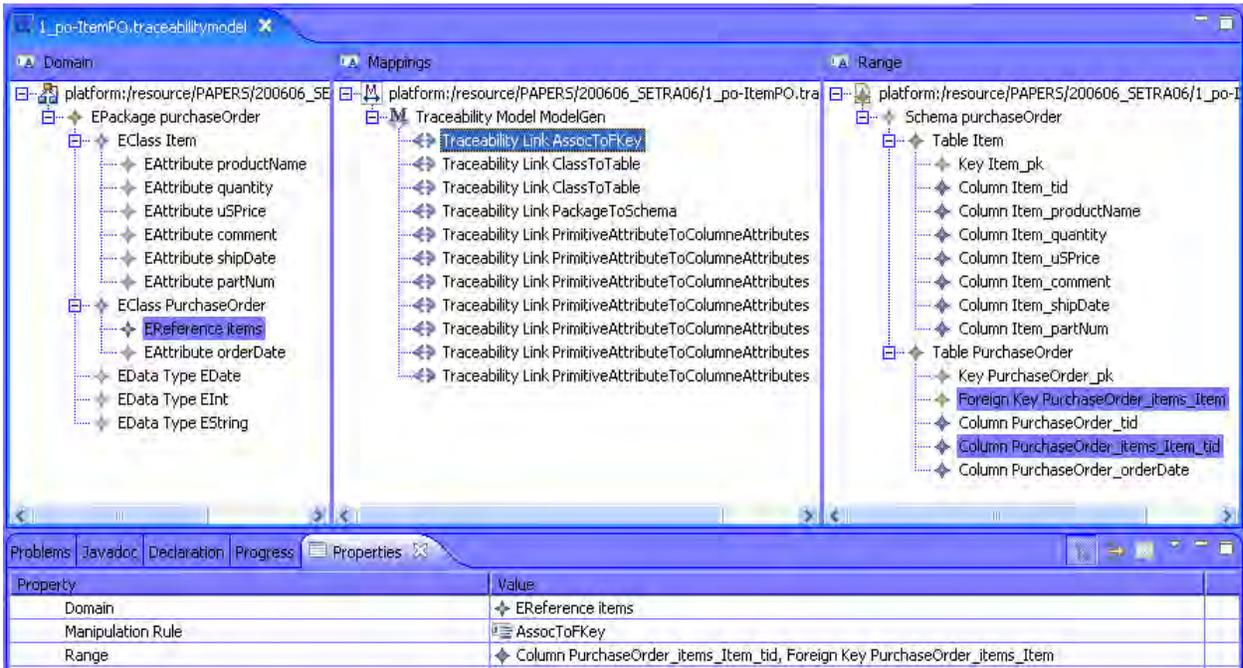


Figure 2: Traceability Editor in the MOMENT Framework.

applied to each source element. A QVT Relations enforced transformation is executed by means of the *ModelGen* operator as follows:

$$\langle output_model, trac_1, \dots, trac_n \rangle = ModelGen(transformation, input_model_1, \dots, input_model_n)$$

where *transformation* is the name of the QVT transformation; *input_model₁*, ..., *input_model_n* are the input models, which may conform to different metamodels; *output_model* is the generated model; and *trac₁*, ..., *trac_n* are the trace models that are generated for each one of the corresponding input models.

Fig. 2 presents the traceability editor of the MOMENT framework. This editor shows the trace model that is generated by the *UmlToRdbms* transformation, when it is applied to the UML model that is defined in Fig. 1. This transformation constitutes the first step of the exogenous model merging process. Trace models in our framework conform to our traceability metamodel, which was presented in [BCR05]. The traceability editor is constituted by three main frames, the left frame shows an input model of the transformation, the right frame shows the output generated model and the frame in the middle shows the traces that relate elements of the input model to elements of the target model. Traces also provide information about the transformation rule (or relation) that has been applied to source elements to generate the corresponding trace and the related target elements.



4 The *Merge* Operator

The *Merge* operator takes two models as input and produces a third one. If A and B are models that conform to the same metamodel, the application of the *Merge* operator on them produces a model C, which consists of the members of A together with the members of B, i.e. the union of A and B. Taking into account that duplicates are not allowed in a model, the union is disjoint.

To understand the semantics of the *Merge* operator in our example, we need to introduce two concepts: the equivalence relation, for finding duplicates by comparing models, and the conflict resolution strategy, for integrating them.

4.1 The equivalence relation

In an endogenous model merging, an equivalence relation is defined between elements that belong to different models that conform to the same metamodel. To define an equivalence relation among the elements of a model in our approach, the user can use the QVT Relation language in the checkonly mode. Only checkonly transformations with two domains are accepted in this context. Both domains have to refer to the same metamodel in our approach. For the example, we customize the *Merge* operator to merge relational schemas, i.e., models that conform to the *RDBMS* metamodel of Appendix A. To do so we use a checkonly QVT Transformation whose domains refer to the *RDBMS* metamodel. The user can add a QVT relation for each of the classes that appear in the metamodel when it is desired. Such QVT relations act as equivalence relationships that must hold over the elements of two *RDBMS* models. These QVT relations are used in the merging process to check when two elements are equivalent in order to eliminate duplicates.

For instance, the following relation can be defined to indicate that two tables are the same if they belong to the same schema and they have the same name by means of the *tableName* variable³:

```

top relation TableEquivalence {
  tableName: String;
  checkonly domain rdbmsDomain1 t1: Table {
    schema = s1:Schema {},
    name=tableName
  };
  checkonly domain rdbmsDomain2 t2: Table {
    schema = s2:Schema {},
    name=tableName
  };
  when {
    SchemaEquivalence(s1, s2);
  }
}

```

where the *SchemaEquivalence* is another QVT Relation defined within the same transformation, describing when two Schema instances are equivalent (for instance, by name). In our approach, this kind of equivalences may involve several instances of two models as in the above example, where *Table* instances

³We have chosen these criteria for the example. Nevertheless, they can be customized to a specific metamodel by the user. Nothing impedes us to add semantic annotations to the elements of a model and use this information to determine which elements are equals or not.



and *Schema* instances are used to check whether two tables are equivalent or not.

During the merging process, this checkonly transformation permits checking when groups of elements of different models represent duplicate elements so that they will be merged. In a checkonly QVT transformation, helper functions can be defined by using OCL expressions to manipulate and compare names, and to navigate the structure of the corresponding model. Thus, the user only has to be aware of the standard QVT Relations language and the domain-specific knowledge.

4.2 The conflict resolution strategy

During a model merging process, when two software artifacts (each of which belongs to a different model) are supposed to be equivalent, one of them must be erased. Their syntactical differences may cast doubt on which should be the syntactical structure for the merged element. Here, the conflict resolution strategy comes into play. The conflict resolution strategy is a model transformation that has two input models and one output model, the merged one. The generic semantics of this strategy in our framework consists of the preferred model strategy. When the *Merge* operator is applied to two models, one has to be chosen as preferred (the first argument of the *Merge* operator). In this way, when two groups of elements (that belong to different models) are equivalent due to an equivalence relation, the elements of the preferred model prevail although they may differ syntactically.

To refine the *Merge* operator, the conflict resolution strategy can also be customized. During the merging process, when the *Merge* operator finds two duplicates, they should be integrated. This integration involves a transformation process where information of both duplicates may be taken into account to define the merged model. Thus, an enforced QVT transformation can be used to customize the conflict resolution strategy in the same way a checkonly QVT transformation is used to customize the generic equivalence relation.

A QVT transformation that is used to define a specific conflict resolution strategy has three domains. All of them refer to the metamodel under study (*RDBMS* in our example). The first two domains are defined as checkonly and they only query the two input models of the *Merge* operator. The third domain is defined as enforce and is the one that produces merged elements. In the case study, when we integrate two tables that are equivalent (because they have the same name), we have to integrate their respective columns, primary keys and foreign keys. The following QVT Relation is intended to perform this task:

```

top relation TableMerging {
  tableName: String;
  checkonly domain rdbmsDomain1 t1: Table {
    schema = s1:Schema {},
    name = tableName
  };
  checkonly domain rdbmsDomain2 t2: Table {
    schema = s2:Schema {},
    name = tableName
  };
  enforce domain rdbmsDomain3 t3: Table {
    schema = s3:Schema {},
    name = tableName
  };
  when {

```



```

    SchemaMerging(s1, s2, s3);
  }
  where {
    ColumnMerging(t1, t2, t3);
    PKMerging(t1, t2, t3);
    FKMerging(t1, t2, t3);
  }
}

```

where the *SchemaMerging* QVT relation, which is invoked in the *when* clause, ensures that the container schemas of both *Table* instances must be equivalent in order to apply the current relation to the involved tables. The QVT Relations that are invoked in the *where* clause ensure that the merging process will go on by merging columns, primary keys and foreign keys of the involved tables.

The enforce QVT transformation that the user defines to customize the conflict resolution strategy is automatically compiled into a *ModelGen* equation as briefly introduced in the previous section ⁴.

4.3 The Merge operator

The *Merge* operator takes two models that conform to the same metamodel as inputs. The outputs of the *Merge* operator are a merged model (*merged_model*) and two models of traces (*trac₁* and *trac₂*) that relate the elements of each input model (*model1* and *model2*) to the elements of the output merged model. The operator is used as follows:

$$\langle merged_model, trac_1, trac_2 \rangle = Merge(model1, model2)$$

The *Merge* operator uses the equivalence relation that is defined for a metamodel to detect duplicated elements between the two input models. When two duplicated elements are found, the conflict resolution strategy is applied to them in order to obtain merged elements, which are then added to the output model. The elements that belong to only one model, without being duplicated in the other one, are copied into the merged model.

The two output trace models are automatically generated by the *Merge* operator on the grounds that it reuses the model transformation mechanism that is described in Section 3, through the conflict resolution strategy. These trace models provide full support for keeping traceability between the input models and the new merged one. The second step of the exogenous model merging in the example constitutes a merging process that involves the model *RDBMS'* and the model *RDBMS*. The model *RDBMS'* is the result of applying the *UmlToRdbms* transformation (defined in Appendix B) to the model *UML* that is defined in Fig. 1, as explained in Section 3. The model *RDBMS* is provided in Fig. 1. In Fig. 3, we show the trace model that is generated during this merging process for the *RDBMS'* model (shown in the left frame of the editor). The model that appears in the right frame of the editor is the final merged relational schema.

⁴More information about the semantics of the *Merge* operator can be found in [BCRL06]

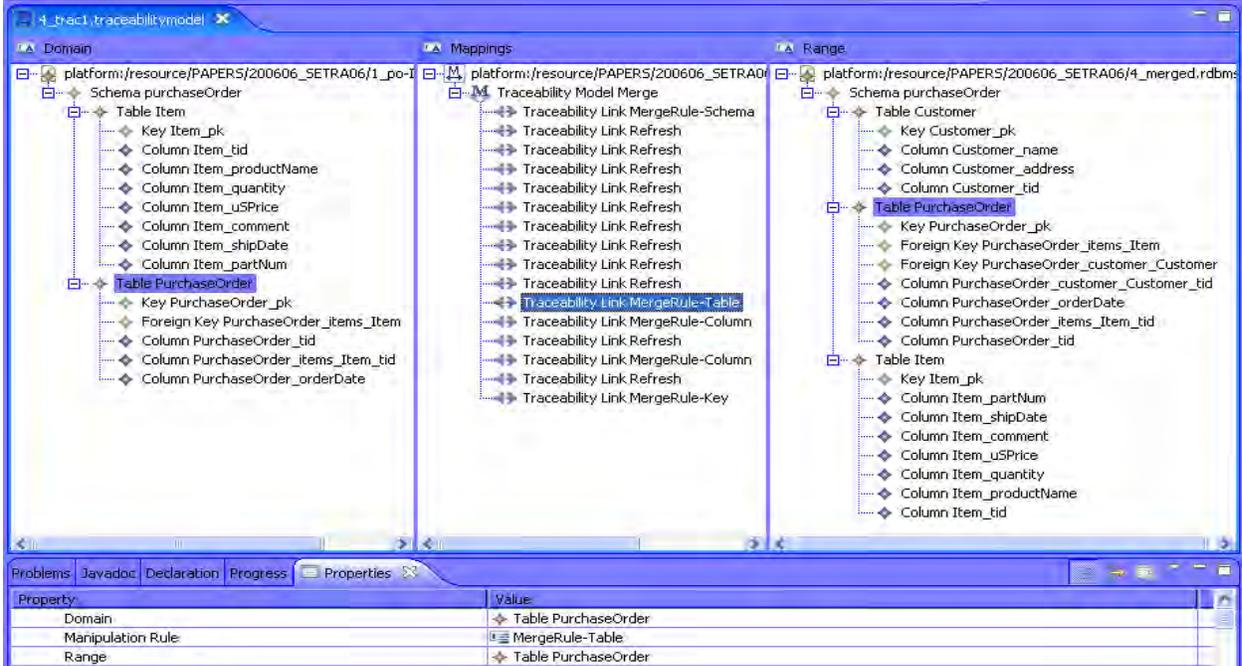


Figure 3: Trace model that is produced during the merging of the models *RDBMS'* and *RDBMS*.

5 Exogenous Model Merging in MOMENT

The exogenous model merging problem consists in the merging of two models that conform to different metamodels, as in the example in Section 2. This problem can be divided into simpler ones that can be solved by two simple model management operators. A composite operator, called *ExogenousMerge*, can be defined for this purpose by composing the *Merge* operator and the *ModelGen* operator. This operator has three arguments: the model A, which conforms to the metamodel *MMA* (the *Ecore* metamodel in our example); the model B, which conforms to the metamodel *MMB* (the *RDBMS* metamodel in our example); and the name of the QVT transformation that must be defined between the metamodels *MMA* and *MMB* (*umlToRdbms* in our example). In the first step, model A is transformed into a model B', which conforms to the metamodel *MMB* by means of the operator *ModelGen*. This step has been performed in Section 3. In the second step, models B and B' are merged within the metamodel *MMB*. This step has been performed in Section 4. Finally, the merged model *result* is the output of the composite operator. The definition of the *ExogenousMerge* composite operator is as follows:

$$\text{operator } ExogenousMerge (A : MMA, B : MMB, T : Transformation) =$$

$$\langle B', \text{map}_{A \rightarrow B'} \rangle = ModelGen(T, A) \quad (1)$$

$$\langle \text{result}, \text{map}_{A \rightarrow B'}, \text{map}_{A \rightarrow B} \rangle = Merge(B', B) \quad (2)$$

return (result)



The *ExogenousMerge* operator is defined independently of any metamodel so that it can be reused to merge two models that conform to any metamodel. In this example, we have not taken into account the trace models that are generated by the *ModelGen* and *Merge* operators. Nevertheless, another version of the operator could generate traceability models as result of the *ExogenousMerge* operator.

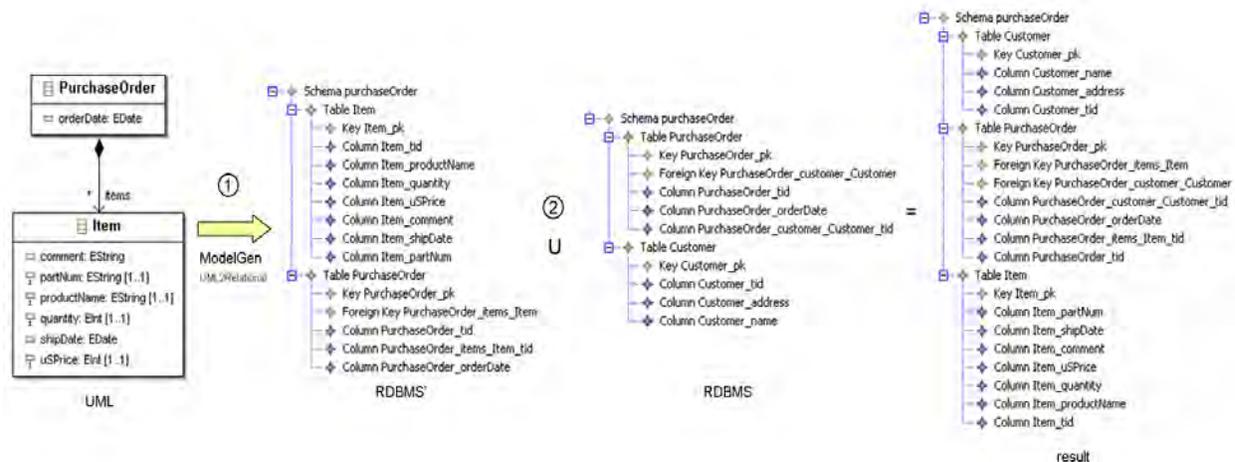


Figure 4: Application of the *ExogenousMerge* operator to the example in Section 2.

Fig. 4 graphically represents the merging process that is performed by the operator *ExogenousMerge* for solving the example that is shown in Section 2. In the example, parameter A corresponds to the UML model and parameter B corresponds to the RDBMS model in Fig. 4. To be able to apply the operator, the equivalence relation for the RDBMS metamodel and the transformation function between the UML and the Relational metamodels must be previously defined by the user.

6 Related Work

Generic model merging approaches take into account the phases that were discussed in [BLN86] to merge database schemas. These approaches can be differentiated by the mechanism that is used to perform model comparison.

[AP03] uses MOF identifiers to compare elements in different versions of a same base model. Although this approach is effective, only versions of a same base model can be compared and merged.

[BP03, BCE⁺06] provide a set of model management operators to define equivalence relationships between elements of different models by means of mappings, which are used by a merge operator later on. In this approach, the *Merge* operator receives two models (A and B) and a mapping model (mapAB) between them as inputs, and it produces the merged model C and two new mapping models (*mapAC* and *mapBC*): $\langle C, mapAC, mapBC \rangle = Merge(A, B, mapAB)$.

In the AMMA platform [FJ05], the Generic Model Weaver AMW is a tool that permits the definition of mapping models (called weaving models) between MOF models in the ATLAS Model Management



Architecture. AMW provides a basic weaving metamodel that can be extended to permit the definition of complex mappings. These mappings are usually defined by the user, although they may be inferred by means of heuristics, as in [MBR01]. These mapping models are used, together with the mapped models in a model transformation to perform a model composition.

In MOMENT, mapping models are introduced as trace models that are generated by model management operators. This is because operators do not have to rely on them to be applied to a set of models. In MOMENT, mappings between the elements of two models are defined between the elements of their corresponding metamodels by means of checkonly QVT Relations. This permits a clearer specification of composite operators. Trace models are produced by the application of a simple operator to a set of models and keep information about the manipulation task that has been performed to a model.

[KPP06] proposes several domain-specific languages to define model comparison and model merging over metamodels. The model comparison language enhances the definition of equivalence relationships between elements of a metamodel that can be applied over the elements of the corresponding models afterwards. In this language, a differentiation between matching and conformance is provided. While a matching mapping indicates when two elements are equivalent, a conformance mapping indicates when two elements are equivalent and consistent to be merged. In this approach, when an equivalence relationship based on names is used, two elements do not conform to each other if they have different types, for instance. In our approach, we use the QVT Relations language to perform model comparison and model transformation. This feature aims at decreasing the learning curve of our framework since there is only one language, which has been specified as a standard. The QVT Relation language does not provide such a differentiation between conformance and matching. Since two elements that do not conform to each other are usually interpreted as an error, we collapse the conformance and matching conditions in a relation. However, a transformation with conformance relations could be defined for a specific metamodel. Then, this transformation could be specialized with user-defined checkonly relations for defining equivalence relationships.

7 Conclusions

Model merging plays a relevant role in the maintenance and evolution of model-based software. Systems of this kind are usually represented by models that conform to different metamodels. Thus, two kinds of merging processes arise by considering the amount of metamodels that are involved: endogenous merging and exogenous merging. In an endogenous merging process, the models that are merged conform to the same metamodel. In an exogenous merging process, the models that are merged conform to different metamodels.

The MOMENT framework is a model management framework that provides operators to manipulate models on top of a MOF architecture, such as *Merge* for model merging and *ModelGen* for model transformations. In our approach, model management operators are defined independently of any metamodel, keeping a generic infrastructure, but they might be customized by an expert user with domain-specific knowledge by means of standard languages, such as OCL and QVT.

In this paper, we have presented how model transformations are supported in MOMENT through the QVT Relations language and how model transformations play an important role in a model merging



process. We have used the standard QVT Relations for this purpose instead of providing new languages for model comparison and model merging. To study the aforementioned kinds of model merging, we have described a solution for an endogenous model merging process by using model transformations through the *Merge* operator. Finally, we have provided a generic solution for exogenous model merging by reusing model transformations and endogenous model merging.

8 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.

We are grateful to Abel Gómez, Pascual Queral, Joaquín Oriente and Luis Hoyos for their effort in the development of the MOMENT Framework.

References

- [AP03] Marcus Alanen and Ivan Porres. Difference and union of models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 2–17. Springer, 2003.
- [BBM03] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.
- [BCE⁺06] Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad Sabetzadeh. A manifesto for model merging. In *GaMMA '06: Proceedings of the 2006 international workshop on Global integrated model management*, pages 5–12, New York, NY, USA, 2006. ACM Press.
- [BCR05] Artur Boronat, José A. Carsí, and Isidro Ramos. Automatic support for traceability in a generic model management framework. In Alan Hartman and David Kreische, editors, *Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005*, volume 3748 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2005.
- [BCRL06] Artur Boronat, José A. Carsí, Isidro Ramos, and Patricio Letelier. Formal model merging applied to class diagram integration. *Electr. Notes Theor. Comput. Sci. (Accepted for publication)*, 2006.
- [BDK92] Peter Buneman, Susan B. Davidson, and Anthony Kosky. Theoretical aspects of schema merging. In *Extending Database Technology*, pages 152–167, 1992.
- [Beh00] Ralf Behrens. A grammar based model for XML schema integration. *Lecture Notes in Computer Science*, 1832:172+, 2000.



- [Ber03] Philip A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.
- [BHP00] Phillip A. Bernstein, Alon Y. Halevy, and Rachel A. Pottinger. A vision for management of complex models. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(4):55–63, 2000.
- [BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.*, 18(4):323–364, 1986.
- [BP03] Philip A. Bernstein and Rachel A. Pottinger. Merging models based on given correspondences. In *Proceedings of the 29th VLDB Conference*, Berlin, 2003.
- [FJ05] Marcos Didonet Del Fabro and Frédéric Jouault. Model transformation and weaving in the amma platform. In *Pre-proceedings of the Generative and Transformational Techniques in Software Engineering (GTTSE'05), Workshop*, pages 71–77, Braga, Portugal, 2005. Centro de Ciências e Tecnologias de Computação, Departamento de Informatica, Universidade do Minho.
- [HM05] Peter Haase and Boris Motik. A mapping system for the integration of owl-dl ontologies. In *IHIS '05: Proceedings of the first international workshop on Interoperability of heterogeneous information systems*, pages 9–16, New York, NY, USA, 2005. ACM Press.
- [KPP06] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model comparison: a foundation for model composition and model transformation testing. In *GaMMA '06: Proceedings of the 2006 international workshop on Global integrated model management*, pages 13–20, New York, NY, USA, 2006. ACM Press.
- [MBR01] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching using cupid. In *Proc. VLDB 2001*, pages 49–58, 2001.
- [Men02] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.
- [OMG04] Object Management Group OMG. Meta object facility (mof) 2.0 core specification (ptc/04-10-15). <http://www.omg.org/cgi-bin/doc?formal/2006-01-01>, 2004.
- [OMG05] Object Management Group OMG. Mof 2.0 qvt final adopted specification (ptc/05-11-01). <http://www.omg.org/cgi-bin/doc?ptc/2005-11-01>. 2005.
- [OWK03] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of uml diagrams. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 227–236, New York, NY, USA, 2003. ACM Press.



- [SGS⁺04] Greg Straw, Geri Georg, Eunjee Song, Sudipto Ghosh, Robert B. France, and James M. Bieman. Model composition directives. In Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors, *UML*, volume 3273 of *Lecture Notes in Computer Science*, pages 84–97. Springer, 2004.

A RDBMS Metamodel

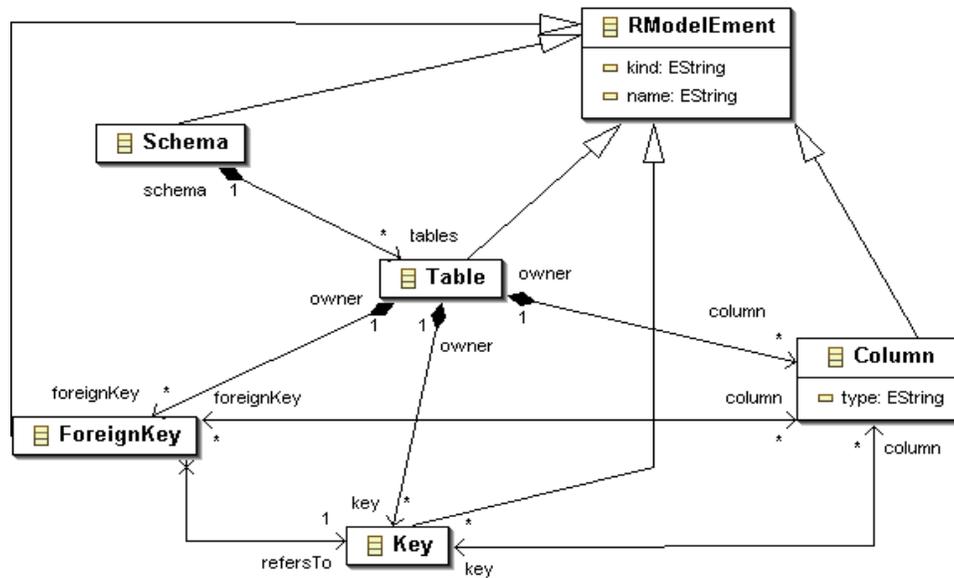


Figure 5: RDBMS metamodel.

B An Ecore to RDBMS Transformation by means of the QVT Relations Language

```

transformation umlToRdbms(ecoreDomain:ecore, rdbmsDomain:rdbms) {
  key Schema {name};
  key Table {schema,name};
  key Column {owner,name};
  key ForeignKey {owner,name};

  top relation PackageToSchema {
    packageName: String;
    checkonly domain ecoreDomain p:EPackage {
      name=packageName
    };
    enforce domain rdbmsDomain s:Schema {
      name=packageName
    };
  }//end PackageToSchema
}
  
```



```

top relation ClassToTable {
  className: String;
  checkonly domain ecoreDomain c: EClass {
    ePackage = p:EPackage {},
    name=className
  };
  enforce domain rdbmsDomain t: Table {
    schema = s:Schema {},
    name = className,
    column = cl:Column {
      name = className + '_tid',
      type = 'NUMBER'
    },
    key = k:Key {
      name = className + '_pk',
      column=cl
    }
  };
  when {
    PackageToSchema(p, s);
  }
  where {
    AttributeToColumn(c, t, className);
  }
} //end ClassToTable

relation AttributeToColumn
{
  checkonly domain ecoreDomain c:EClass {};
  checkonly domain rdbmsDomain t:Table {};
  primitive domain prefix:String;
  where {
    PrimitiveAttributeToColumn(c, t, prefix);
    SuperAttributeToColumn(c, t, prefix);
  }
} //end AttributeToColumn

relation PrimitiveAttributeToColumn
{
  attributeName, columnName, sqltype: String;
  checkonly domain ecoreDomain c:EClass {
    eAttributes = a:EAttribute {}
  }
}

```



```

};
checkonly domain rdbmsDomain t:Table {};
primitive domain prefix:String;
where {
    PrimitiveAttributeToColumnAttributes(a,t,prefix);
}
} //end PrimitiveAttributeToColumn

relation PrimitiveAttributeToColumnAttributes
{
    attributeName, columnName, ecoreTypeName, sqltype: String;
    checkonly domain ecoreDomain a:EAttribute {
        name = attributeName,
        eType = ecoretype: EDataType {
            name = ecoreTypeName
        }
    };
    checkonly domain rdbmsDomain t:Table{};
    enforce domain rdbmsDomain cl:Column {
        name = (
            if (prefix = "") then
                attributeName
            else
                prefix + '_' + attributeName
            endif
        ),
        type = PrimitiveTypeToSqlType(ecoreTypeName),
        owner = t
    };
    primitive domain prefix:String;
    when {
        IsPrimitiveDatatype(ecoreTypeName);
    }
} //end relation

relation SuperAttributeToColumn
{
    checkonly domain ecoreDomain c: EClass {
        eSuperTypes = sc:EClass {}
    };
    checkonly domain rdbmsDomain t:Table {};
    primitive domain prefix: String;

```



```

    where {
      AttributeToColumn(sc, t, prefix);
    }
  }
top relation AssocToFKey
{
  srcTbl, destTbl: Table;
  pKey: Key;
  referenceName, sourceClassName, targetClassName: String;
  checkonly domain ecoreDomain ref: EReference {
    name = referenceName,
    eContainingClass = sc:EClass {
      name = sourceClassName
    },
    eType = tc:EClass {
      name = targetClassName
    }
  };
  enforce domain rdbmsDomain fk:ForeignKey {
    name = sourceClassName + '_' + referenceName + '_' + targetClassName,
    owner = srcTbl,
    column = fkc:Column {
      name = sourceClassName + '_' + referenceName + '_' + targetClassName + '_tid',
      type = 'NUMBER',
      owner = srcTbl
    },
    refersTo = ObtainReferredPrimaryKey(destTbl)
  };
  when {
    ClassToTable(sc, srcTbl);
    ClassToTable(tc, destTbl);
  }
}
function ObtainReferredPrimaryKey(table: Table):Key
{
  table.key
}
function IsPrimitiveDatatype(datatype: String):Bool
{
  ((datatype = 'EInt') or (datatype = 'EBoolean') or (datatype = 'EString') or (datatype = 'EDate'))
}

```



```
}  
function PrimitiveTypeToSqlType(primitiveType:String):String  
{  
  if (primitiveType='EInt')  
    then 'NUMBER'  
  else if (primitiveType='EBoolean')  
    then 'BOOLEAN'  
  else  
    'VARCHAR'  
  endif  
endif  
}  
function IsDirectedReference(ref:EReference):Bool  
{  
  (ref.eOpposite -> size() = 0)  
}  
}
```




An Algorithm for Detecting and Removing Clones in Java Code

Nicolas Juillerat and Béat Hirsbrunner *

*University of Fribourg, Department of Computer Science, 1700 Fribourg, Switzerland

***Abstract.** This paper proposes a new algorithm for automatically detecting and removing duplicated code in existing Java programs. Its purpose is to improve the structure of small code snippets (as in refactoring), rather than reducing the overall redundancy in huge legacy programs. As such, approaches that favor code clarity over efficiency are introduced. The skeleton of our algorithm is presented and illustrated on concrete examples of code.*

Keywords: clone detection, clone removal, code duplication, extract method, refactoring, Java

1 Introduction

As a big software project grows, code duplication is a common problem. When the developer team is under the pressure of the deadline, and when the specification is constantly changing, the programmers frequently just copy and paste existing code, and apply a few (sometimes empty) modifications. This results in code duplication, or so called “clones” which makes maintenance and debugging very difficult [9, 13].

The process of detecting and removing clones consists of two steps: detecting the duplicated statements first, and then extracting them into new methods and replacing their occurrences by invocations to the corresponding methods. In this paper, we propose a new algorithm for both steps.

While existing algorithms are mainly targeted to the maintenance of big legacy programs [5, 11] in various languages, our solution is targeted to small Java code snippets, and is meant to be used like any other refactoring tool, in a semi-automated way. As such, our algorithm proposes new solutions that favor the clarity of the resulting code rather than the overall performance.

This paper assumes the reader has knowledge of the Java language and of the notion of abstract syntax tree (AST). A large part of the presented algorithm consists of an implementation of the *extract method* refactoring. Because this transformation has already been covered by previous papers [1, 3, 4, 6], we give the focus on the remaining parts only.

This paper is organized as follow: section 2 gives an overview of existing and related work, in order to define the context of this paper and its contribution over previous work. Section 3 presents an algo-



rithm for the detection of clones. Section 4 discusses the problem of removing the detected clones, and presents a generic solution for the Java language that can easily be extended and adapted to other similar programming languages. The full algorithm is summarized in section 5, and its limitations and possible improvements are stated in section 6.

2 Related work

There are multiple ways of *detecting* clones, including string based and token based solutions. An overview of existing techniques is given by F. Von Rysselberghe and S. Demeyer [9]. While string based techniques are possible, more promising techniques are those based on the Program Dependency Graph [11] and on the AST [13]. Our solution is based on the AST and has thus some similarities with the solution of Ira D. Baxter [13].

The problem of *extracting* the detected clones has been covered by various papers [8, 12, 13, 16]. Removing a clone consists in extracting it into a new method, and replacing all its occurrences by invocations to that method. This is a complex problem compared to the clone detection because it is constrained by the preconditions of the *extract method* refactoring [1, 4, 6].

W. Griswold and D. Notkin [16] cover this problem on the Scheme language and Ira D. Baxter [13] on the C language with the help of macros. In both cases the process is greatly simplified by the choice of the target programming language. S. Horwitz [12] gives a solution for the C language (without using macros), but does not handle jumps. The solution is refined in a next paper [8] to handle jumps as well and to perform clone extraction even in “difficult” cases. Unfortunately it relies on the possibility of passing arguments by reference, which is not possible in the Java language.

The solution presented in this paper is mainly targeted to the Java language. We give the following new contributions over previous work:

- Unlike previous solutions, we do not make use of code transformations before the clone extraction such as promotion, statement reordering, predicate duplication or jump transformations [8]. We instead allow a clone to be extracted into more than one method if necessary. As a result, the transformed code remains closer to the original one.
- Regarding the clone removal, our solution properly handles the limitations imposed by the Java language that cannot pass parameters by reference. The proposed solution does not require any artificial tricks for that purpose, which would greatly reduce the clarity of the code. Note that most existing solutions for the Java language are limited to clone *detection* and simply do not handle clone *removal* [5, 9].
- Previous work are based on clones consisting of statements that are consecutive (or eventually disjoint but close to each other). Our solution not only works for consecutive statements, but also detects and removes small clones hidden in *sub-expressions*.

These choices can reduce the quality of the result in term of the overall reduction of duplications, compared to other existing solutions. Nevertheless they are justified by the use of our algorithm as a *refactoring tool*: in this context, the clarity of the resulting code and its fidelity to the original one are the main

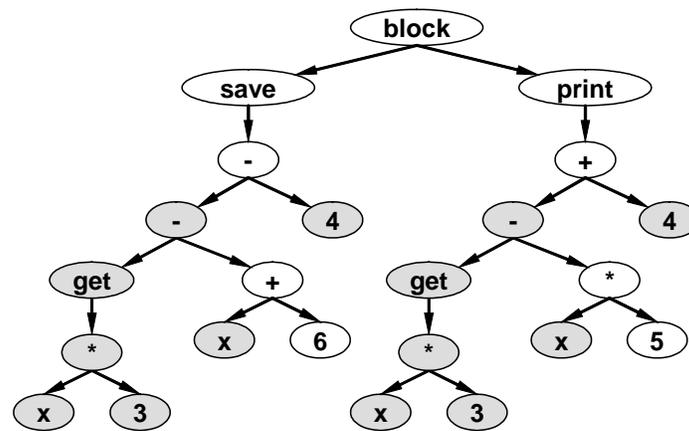


Figure 1: The AST of the two Java statements. Nodes that are part of a clone are shaded.

concerns [10]. Refactorings dealing with small sub-expressions exist [6] and hence detecting clones in sub-expressions may be useful as well in this context.

3 Clone detection

In this section we present the first part of the process: clone detection.

Our solution for the detection of clones is based on the abstract syntax tree (AST). The first step consists of converting the AST into a list of tokens using a post-order walk of the tree. Consider the following Java statements as an example:

```
save(get(x * 3) - (x + 6) - 4);
print(get(x * 3) - (x * 5) + 4);
```

The figure 1 shows a possible AST of these statements. Note that we have chosen a representation in which consecutive statements belonging to the same block are children of a **block** node. The algorithm we are discussing can easily be adapted to other representations.

A post-order tree walk of the AST results in the following list of tokens:

```
[x, 3, *, get, x, 6, +, -, 4, -, save, x, 3, *, get, x,
5, *, -, 4, +, print, block]
```

This list of tokens exhibits some duplication which are also visible in the source code and in the AST. But it has some advantages over the two other representations: like the AST, it is abstract and hence does not contain any element such as white spaces or comments which are not relevant. But unlike the AST, the fact it is a list allows us to use the variety of existing tools for the detection of duplications in lists.

For instance, if we want to detect duplications between two similar methods, a differentiation algorithm, such as the one used in the Linux **diff** command [18], can be used on the two token lists. In the



more general case, we can use lossless data compression algorithms (usually based on hashing), such as the LZ77 algorithm [14, 15]: the first step of this lossless data compression algorithm and its variants precisely consists in detecting clones in a list of tokens. Other methods are covered in the literature [9].

With these techniques, the two following duplicated sublists of tokens are detected in our previous example:

```
[x, 3, *, get, x], [-, 4]
```

These two sublists seem to correspond to clones. Unfortunately we cannot safely extract them into methods: the *extract method* refactoring has various preconditions [2] and at least one of them is not satisfied: in both cases, the sublist does not correspond to a single expression or to a list of consecutive expressions or statements. More generally, we cannot just expect that a clone will satisfy all the preconditions that are required to extract it: the *extract method* refactoring has a lot of preconditions compared to other transformations [6].

Rather than modifying the first step of our algorithm, we will now introduce a second step, whose purpose is precisely to satisfy all the preconditions of the *extract method* refactoring and to correct some side-effects of the post-order tree walk used for the detection of clones.

4 Removing clones

The previous section described a way of detecting duplicated code fragments. Once these clones are detected, they cannot always be extracted into methods immediately. The solution we propose is to apply various *constraints* on the previous result. The purpose of each constraint is to modify the detected clones in such a way they can safely be extracted. This section explains this idea by illustrating these constraints, most of which correspond to the preconditions of the *extract method* refactoring.

4.1 Splitting expressions

Coming back to the example of the previous section, one of the detected sublists of similar tokens was:

```
[x, 3, *, get, x]
```

These tokens correspond to two subtrees on the AST: the first one is formed by the first four tokens, and the second one is formed by a single token, the last one. In general, a sublist of tokens forms a *forest* of one or more subtrees of the AST. In our case, because the sublist corresponds to two subtrees, it also corresponds to two sub-expressions. Thus, it is not possible to build a single method which returns two values.

Therefore, the first constraint we have to apply is to split the detected sublists in such a way each of them corresponds to a single subtree and hence to a single expression. This is feasible if the tokens of the lists are linked to the corresponding nodes of the AST.

Note that this constraint is only necessary for subtrees whose parent nodes are not the same **block** node. If two or more consecutive subtrees are children of the same **block** node, they correspond to consecutive *statements* and can be kept together.



Applying this constraint to the result of the previous section produces four sublists of tokens corresponding to four sub-expressions:

```
[x, 3, *, get], [x], [-], [4]
```

Note that the third sublist corresponds to a single sub-expression (a subtraction), although the operands are not included. Extracting four methods corresponding to these four sub-expressions results in the following code¹:

```
save(sub(expr1(), (getX() + 6)) - get4());
print(sub(expr1(), (getX() * 5)) + get4());
```

where `expr1` is a function returning “get (x * 3)”, `sub` is a function subtracting its two arguments, `getX` is a function that just returns `x` and `get4` is a function that returns the value 4.

4.2 Applying a threshold

The previous result immediately suggests an obvious new constraint: applying a threshold. More precisely, sublists of similar tokens whose lengths are below a threshold are probably not worth putting in a new method. By dropping the three sublists consisting of only one token from the previous example, the resulting code looks much better:

```
save(expr1() - (x + 6) - 4);
print(expr1() - (x * 5) + 4);
```

The result is the expected one: we have successfully removed the following cloned sub-expression: “get (x * 3)”.

4.3 Control statements

To handle control statements, that is, statements having bodies such as **if**, **while** or **for**, a new constraint is necessary: if it is part of a clone, the statement itself must be dropped from the corresponding list of tokens unless its body is fully included in the clone. The reason is that such a statement cannot be extracted without its entire body [1, 4, 8]. When the token is dropped, the initial list of tokens is eventually split in two lists, resulting in two methods to extract.

Most other constraints imposed by control statements are already handled by the expression splitting process described in section 4.1. As an example, consider that we have the following list of tokens corresponding to the AST shown in figure 2:

```
[cond, stmt1, stmt2, stmt3, block, if, stmt4, block]
```

And assume that two clones corresponding to the two following sublists have been found:

¹In practice, an automatic tool would generate arbitrary method names. We use more meaningful names only for the sake of clarity.

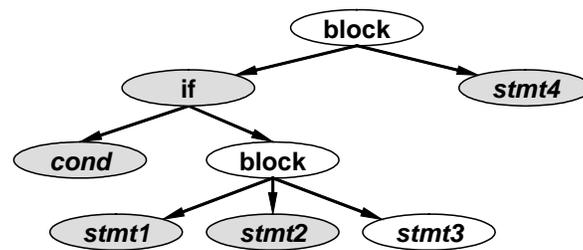


Figure 2: An AST with a clone that only partially covers the body of an **if** statement. The nodes that are part of a clone are shaded. “*cond*” is an arbitrary expression’s subtree and “*stmt1*” to “*stmt4*” are arbitrary statements’ subtrees.

`[cond, stmt1, stmt2], [if, stmt4]`

Because the body of the **if** statement is not fully included in the second clone, the corresponding token must be dropped from the second sublist. The expression splitting process described in section 4.1 will further separate the condition from the first two body statements in the first sublist.

As a result, three clones to extract are produced: one for the condition, one for the first two body statements “*stmt1*” and “*stmt2*”, and one for the “*stmt4*” statement. These clones can all be extracted safely.

More formally, when dealing with statements that have bodies, a method can only be extracted if it crosses no or both boundaries of the statement’s body. The other cases, where only one boundary of the body is crossed, are handled in the following ways by our algorithm:

- Statements before the body and from the begin of the body: these two parts belongs to two different subtrees of the AST. As a result they are extracted in two different methods, as explained in section 4.1.
- Statements from the end of the body and after the body: these two parts are separated by a token corresponding to the control statement itself in the list of tokens. This is a property of the post-order walk used by our algorithm. Removing this token as explained in this section will split the list, and result in two different methods to be extracted.
- Statements including part of a control statement’s expression: unless the whole control statement is covered, the expression splitting process described in section 4.1 will separate the control statement’s expression from the rest, if any.

4.4 Multiple outgoing data flows

A method cannot return more than one value in Java. Therefore we cannot extract a code fragment with multiple outgoing data flows, that is, a code fragment that modifies the value of more than one variable that are read afterwards. As an illustration, consider the following code:



```
min = x - getWindowSize() / 2;
max = x + getWindowSize() / 2;
middle = x;
doStuff(min, middle, max);
```

And assume that the three first statements are duplicated somewhere else in the code, and all the variables are *local* variables and not instance variables. These three statements correspond to the following list of tokens:

```
[min, x, getWindowSize, 2, /, -, =, max, x,
  getWindowSize, 2, /, +, =, middle, x, =]
```

Unfortunately, we cannot extract these statements in a method, because the method would then need to return three values: min, max and middle².

Also note that the constraint presented in section 4.1 will not perform any changes there because we have three subtrees having the same parent **block** node.

A way of dealing with this problem is to split the list of tokens in such a way every sublist has at most one outgoing flow and has a maximum size. The first sublist can be found by repeatedly removing the last token until the condition is satisfied and the result corresponds to a whole expression or to a list of statements. Then the process is applied again with the rest of the clone.

In our example, the whole process will split the list in three sublists:

```
[min, x, getWindowSize, 2, /, -, =],
[max, x, getWindowSize, 2, /, +, =],
[middle, x, =]
```

Now, three methods can safely be extracted, resulting in the following code:

```
min = method1(x);
max = method2(x);
middle = method3(x);
doStuff(min, middle, max);
```

It is necessary to make a few observations at this stage. First, identifying the number of outgoing flows in a code fragment is far from being a simple task and requires a complex flow analysis. This process is part of our algorithm, but we will not address it as it is similar to existing implementations that are already heavily covered by the literature [1, 2, 3, 11].

Second, by splitting a list of tokens into smaller sublists, we may fall into resulting sublists that are very short. The threshold constraint presented in section 4.2 may eventually drop some sublists. In extreme cases, this may dramatically reduce the number of methods we can extract. A worst case example (but fortunately unlikely to occur in practice) would be a code fragment where every variable is postfix with the “++” operator, such as:

```
weird(x++ + y++, x++ + y++, x++ + y++);
```

²Note that there would be no problem if these variables were instance variables.



Unless x and y are instance variables, we cannot really extract any duplicated code: the duplicated expression “ $x++ + y++$ ” modifies two variables and cannot be extracted into a method; and the sub-expressions “ $x++$ ”, “ $y++$ ” and “ $+$ ” are too short to be extracted alone.

4.5 Iteration

In the previous section, we assumed that the following code fragment was duplicated and we managed to extract it into common methods:

```
min = x - getWindowSize() / 2;
max = x + getWindowSize() / 2;
middle = x;
```

But there is also some duplication within the fragment itself: the sub-expression

```
getWindowSize() / 2
```

Therefore, to fully remove code duplications, we may need to apply our whole algorithm in a recursive way on the generated methods, until no more clones are found. Note that this is not implemented by a constraint this time, but by a modification of the algorithm itself.

4.6 Multiple outgoing control flows

As described in section 4.4, it is not possible to extract a code fragment with multiple outgoing data flows. This is one of the preconditions of the *extract method* refactoring. Another related precondition is that we cannot extract a code fragment with multiple outgoing control flows either [1, 3, 11].

A code fragment has multiple outgoing control flows when the execution can leave the code fragment at more than one place. This typically occurs when the code fragment contains one or more conditional “jump” statements such as `break`, `continue` or `return`.

Consider the following code as an illustration:

```
boolean stable = false;
while (true) {
    stable = detectClones(code);
    if (stable || code.hasErrors())
        break;
    stable = extractClones();
    if (stable || code.hasErrors())
        break;
}
```

Clearly, there is a duplication of the `if` statement and its body, corresponding to the AST subtree of figure 3, and to the following list of tokens (after a post-order walk):

```
[stable, code, hasErrors, ||, break, if]
```

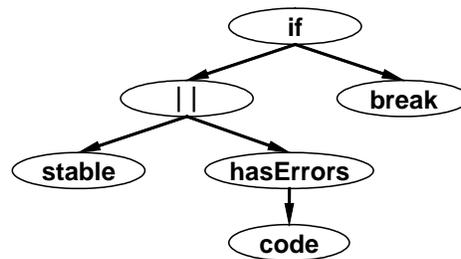


Figure 3: The AST subtree of the duplicated statements.

But the `break` instruction prevents the extraction of the statements corresponding to these tokens: if they were extracted in a new method, this method would contain the `break` statement, but not the enclosing loop, yielding to a compilation error. The problem would be similar with a `return` statement (even when not in a loop): once a `return` statement is extracted, it does no longer return from the original method but from the extracted method, changing the semantics.

The solution we propose is simply to remove the `break` statement from the list of tokens, and to split the list at the position of the `break`. The same process would be performed for a `return` or `continue` statement. In this particular case, we are left with the following two sublists:

```
[stable, code, hasErrors, ||], [if]
```

The second list has a single token, and cannot be extracted for two different reasons explained in sections 4.2 and 4.3. The best we can do is therefore to extract the first sublist, corresponding to the common expression of the two `if` statements.

In more complex cases where the `break` statement occurs in the middle of a big clone, there would also be two additional extracted methods for the statements before and after the `break`.

There are other ways of dealing with these “jump” statements. One solution is presented by S. Horwitz [8]: a single method is extracted with the jump statement. The method must return an additional value which specifies whether the execution has terminated at the end of the method or at the jump statement (actually replaced by a `return` statement). This returned value is then used by the caller to conditionally execute the “real” jump.

Unfortunately, while this alternative solution is feasible for the C language, it will fail most of the time in Java because a method cannot return more than one value: by adding an additional return value, this limitation is easily reached. Also note that this solution, unlike ours, requires code transformations that go beyond the extraction of a method.

5 Summary

At this stage, we can give the following skeleton of our algorithm to detect and remove clones:

- 1 Parse the source code and build the corresponding AST



- 2 Create a list of tokens using a post-order walk on the AST
- 3 Detect clones in the list, using technologies of lossless data compression
- 4 Apply constraints on the produced sublists of tokens corresponding to clones
 - Split unrelated expressions (section 4.1)
 - Resolve multiple outgoing data flows (section 4.4)
 - Resolve multiple outgoing control flows (section 4.6)
 - Handle control statements (section 4.3)
 - Apply a threshold (section 4.2)
- 5 Extract the code corresponding to the remaining sublists of tokens into new methods
- 6 Restart on all the extracted methods until no more duplicates are found

Note that this is only the overall structure of a full-featured algorithm. We have only covered the most important constraints imposed by the Java language; but in practice, there are many other aspects to deal with. These aspects include for example visibility, naming conflicts, anonymous classes and user interface. In fact, each special construct of a given programming language may involve other or different constraints.

An interesting aspect is the *structure* of our algorithm, which exhibits two properties:

- All steps except step 4 are algorithms that are already known and mastered.
- The step 4 consists in the application of a list of independent constraints. Therefore it can be implemented in a clean way involving nearly no coupling (with the exception of some ordering constraints such as applying the threshold last).

In other words, given the already existing algorithms, implementing our algorithm for a given programming language basically consists in identifying and implementing various constraints on lists of tokens corresponding to the clones to extract.

6 Open issues and future works

We have presented the skeleton of an algorithm for the automatic detection and removal of duplicated code. Our solution deals with small clones and focuses on the clarity of the resulting code compared to existing solutions. Nevertheless, some global open issues have to be pointed out.

Our solution can only detect code fragments that are equal according to their representation as an AST. But different statements may have the same semantics although they result in different ASTs. Resolving this problem in general is still an open issue because it would require a lot of semantic analyses and transformations. Some specific cases have been solved though.



- The solution given by Ira D. Baxter [13] for instance can properly detect clones involving commutative operators with the operands swapped; but the extraction of clones for the Java language is not discussed.
- In the paper of F. Von Rysselberghe and S. Demeyer [9], a detection algorithm is presented, that can properly find clones in which variables have been renamed. Unfortunately, the proposed solution can produce false positives, which are not acceptable for the extraction.
- D. Koes et al. [7] discuss the complexity of finding statements which are independent. This task would be necessary if we want to properly detect a code fragment that is duplicated, but where two independent statements are swapped. Possible solutions are given by [8, 11] with the help of the Program Dependency Graph [17].

Another limitation is that our detection algorithm will not always detect “ideal” clones. The clones are often different than those that would be detected manually by a human. This problem was already pointed out for most existing detection algorithms [8, 9].

These issues confirm the importance of the intervention of the programmer during the process. An automated algorithm can be of a great help, but it usually does not give perfect solutions all the times.

7 Conclusion

In this paper, we have presented a new algorithm for the automated detection and removal of duplicated code statements, or clones. Unlike previous work, our solution is meant as a refactoring tool and favors code clarity and fidelity over efficiency.

We have shown that the process of removing the detected clones is mainly constrained by the preconditions of the *extract method* refactoring, which makes it a difficult process. But we have proposed a solution that produces acceptable and correct results despite of these constraints.

Unlike previous work, our solution does not modify the source code before the extraction and also finds clones hidden in sub-expressions. It also correctly handles the limitations imposed by the Java programming languages that do not allow passing parameters by reference, without sacrificing the clarity of the resulting code.

While we gave only the “sketch” of a real and full-featured algorithm, we have shown how such an algorithm could be implemented in a clean and structured way for the Java language.

References

- [1] Mathieu Verbaere, Ran Ettinger and Oege de Moor: *JunGL: a Scripting Language for Refactoring*, 28th International Conference on Software Engineering (to appear), 2006
- [2] Leif Frenzel: *The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs*, Eclipse Magazin, vol. 5, 2006



- [3] Nicolas Juillerat, Béat Hirsbrunner: *FOOD: An intermediate model for automated refactoring*, 5th International Conference on Software Methodologies, Tools and Techniques (to appear), 2006
- [4] IBM Corporation: *Eclipse 3.1 documentation*, 2005, accessible on the home page of Eclipse
- [5] Tom Copeland: *PMD applied*, Centennial Books Online, 2005
- [6] Tom Mens, Tom Tourwé: *A Survey of Software Refactoring*, IEEE Transactions on software engineering, vol. 30, no. 2, pp. 126–139, 2004
- [7] David Koes, Mihai Budiu, Girish Venkataramani: *Programmer Specified Pointer Independence*, MSP'04, ACM, pp. 51–59, 2004
- [8] Raghavan Komondoor, Susan Horwitz: *Effective, Automatic Procedure Extraction*, 11th IEEE International Workshop on Program Comprehension, pp. 33–42, 2003
- [9] Filip Von Rysselberghe, Serge Demeyer: *Evaluating Clone Detection Techniques*, Proceedings of the International Workshop on Evolution of Large Scale Industrial Software Applications, pp. 25–36, 2003
- [10] Martin Fowler: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2002
- [11] Raghavan Komondoor, Susan Horwitz: *Finding Duplicated Code Using Program Dependences*, Lecture Notes in Computer Science, Vol 2028, pp. 383–386, 2001
- [12] Raghavan Komondoor, Susan Horwitz: *Semantics-Preserving Procedure Extraction*, ACM Symposium on Principles of Programming Languages, pp. 155–169, 2000
- [13] Ira D. Baxter et al.: *Clone Detection Using Abstract Syntax Trees*, IEEE Proceedings of ICSM'98, pp. 368–377, 1998
- [14] Mark Nelson, Jean-Loup Gailly: *The Data Compression Book*, M&T Books, 2nd edition, 1995
- [15] Michael Burrows, David Wheeler: *A Block-Sorting Lossless Data Compression Algorithm*, Digital Equipment Corporation, 1994, available on the CiteSeer online library
- [16] W. Griswold, D. Notkin: *Automated Assistance for Program Restructuring*, ACM Transactions on Software Engineering and Methodology, pp. 228–269, 1993
- [17] Jeanne Ferrante, Joe D. Warren: *The Program Dependence Graph and Its Use in Optimizations*, ACM Transactions on Programming Languages and Systems, vol. 9, no. 3, pp. 319–349, 1987
- [18] J. W. Hunt, M. Douglas McIlroy: *An Algorithm for Differential File Comparison*, Bell Laboratories Computing Science Technical Report 41, 1976, available on the home page of the second author



Refactoring Information Systems

- Handling Partial Compositions -

Michael Löwe, Harald König, Michael Peters, and Christoph Schulz
FHDW Hannover, Freundallee 15
D-30173 Hannover, Germany

We present our formal framework for the refactoring of complete information systems, i.e., the data model and the data itself. It is described using general and abstract notions of category theory and can handle addition, renaming and removal of model objects as well as folding and unfolding within complete and partial object compositions.

Keywords: Refactoring, Migration, Graph transformation, Pullback complement

1. Introduction

The only constant thing is change. This is especially true for the information and communication business. Currently, information systems in many companies are subject to change. This is mainly due to the technological progress connected to the Internet which enables completely new sorts of electronic business. Thus, we see big efforts to re-engineer the technical basis on the one hand and to improve the business processes and information models on the other hand [1].

This development has been reflected in the research and development community in the last years. Agile and Extreme Programming Techniques [2] [3] [4] aim at supporting the ongoing reengineering processes by providing refactoring methods, techniques, patterns [5] [6] and tools [7]. These tools enable consistent global changes of a whole software system, for example to introduce some design patterns which are necessary for the system to take the next evolution step. This puts the flexibility into the development process that is needed to keep a system up-to-date (without any over-specifications at the beginning of the development) and to realize changing requirements quickly.

For the time being, agile techniques in database engineering were often restricted to the improvement and change of model artifacts. The main obstacle for agile techniques here is existing data. Attempts to describe semantics-preserving schema transformations that also migrate data can be found in [8] [9]. A transformational approach that considers the instance level is discussed in [10].

If a model of a productive information system is changed, we are faced with one central question: “What shall we do with the data conforming to the old model?” Up to now, we hear two major answers:



1. Leave the data as it is and map the new model to the old one using for example some object-relational-mapping tools [11].¹
2. Migrate the data from the old model to the new one by crafting corresponding migration scripts and performing the (long-running) data migration at night or on the weekend.

Both solutions possess big disadvantages. The first one leads to complex mappings if applied several times. This complexity is very likely to produce performance problems and reduce the development speed of the engineering team in the long run.² The second solution requires long production breaks and consumes a lot of development and test time for software (migration scripts) that is thrown away after success.

We propose another approach, namely the generation of the necessary data migration directly from model refactoring, compare also [12]. One central issue is the *correctness* of the induced migrations. We can only benefit from this approach if we can trust in the produced migrations without any further tests. Therefore, we present a theoretical framework in this paper, which

1. is able to represent models and instances in a uniform meta-model,
2. comes equipped with a suitable notion of model refactoring,
3. provides refactoring-induced correct transformations of the instances (migrations), and
4. proves its applicability by satisfying necessary and natural properties for refactorings and migrations, i.e., that refactoring can be composed in a natural way.

The framework is built on category theory [13] and algebraic graph transformation [14]. In this theory, we not only have a very general notion of structured *object*. By the notion of *morphism*, we also get a natural way of representing (1) typings of instance objects in model objects as well as (2) model changes (refactorings) and instance migrations.

Section 2 presents our current framework built on a double-pullback construction, which can handle addition, renaming, and removal of model objects as well as folding and unfolding within *complete* object compositions [15]. This framework is not able to handle inheritance structures directly. Section 3 provides a slight generalization: We do not longer require that the right-hand side of a migration is a pullback. Instead we re-use the explicit construction of the pullback complements in more general situations. It turns out that this construction enjoys some categorical properties that guarantee uniqueness up to isomorphism. Section 4 shows that the usual sequential composition of refactorings extends to migrations in the generalized framework as well. We sketch in section 5, how the results in this paper can be reformulated on a purely categorical level. We explicitly point out the similarities to the approach of Ehrig et al. using adhesive categories [14]. Section 6 provides a conclusion and contains hints for future research activities.

1 An older and worse version of this approach is: Leave the data-model as it is and redefine the meaning of the data within the model, for example by using comma-separated multi-value fields in a single string column.
 2 The longer this approach is applied, the bigger the problems to switch to the second one.

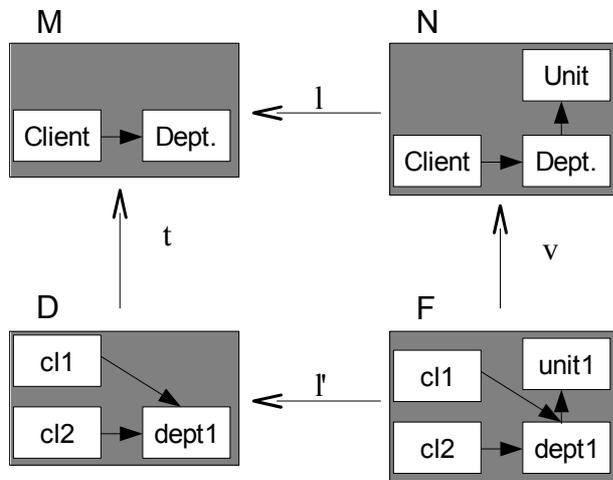


Fig. 1: Extracting a superclass

2. Migration Framework

For a motivation of the following theoretical aspects, consider the situation of a class *Department*. A possible refactoring would be to extract an abstract superclass *Unit* in order to be prepared for additional specializations [5]. If we interpret generalization as object composition on the instance level, an automated migration must add a *Unit*-Object to each *Department*-Object in a 1:1 fashion³. After the migration client objects no longer use a single *Department*-Object but a new object which contains the *Unit*-information as an aggregated object, see Fig. 1. Since *Unit* is an abstract class, we can model this refactoring

by unfolding *Department* to two classes. This can be done by a morphism l that maps the new model N to the old model M assigning the two classes *Department* and *Unit* in N to *Department* in M . Having data D which is typed by the morphism $t: D \rightarrow M$ we obviously can generate the migrated data by calculating the pullback object F of t and l' .

Another possible refactoring is the addition of a new class [5]. This can be achieved with a (non-surjective) map r from the old to the new model, see Fig. 2. Here the question arises which categorical construct generates a reasonable migration. Moreover, different data structures are possible after the migration: one possibility would be to create no B -object, another to create a default-value or prototype object for B . Both solutions lead to pullback diagrams, if objects $a1$ and $a2$ are preserved.

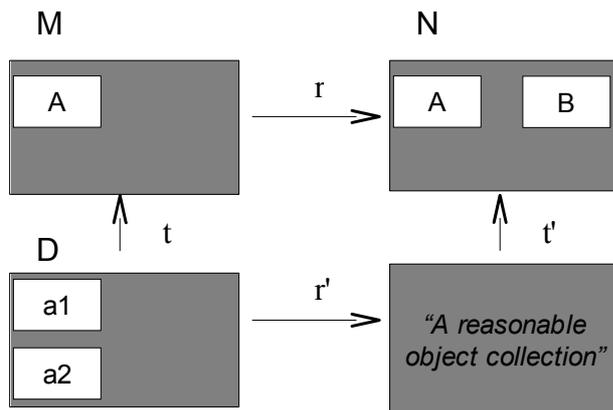


Fig. 2: Adding a new class

Category theory can be applied in the following way. We can express the typing of some data D in a model M by a morphism $t: D \rightarrow M$. And we need to express refactorings between models and migrations between typed data. We will have to use the two variations l and r discussed above. But we are not only interested in the model states before and after refactoring but in the refactoring process itself. Hence, it is a good choice to represent *one* model refactoring from model M to N by a

3 This interpretation is often used when object models are mapped to relational database systems using the “one table per class”-strategy. This strategy provides one relational table for each class and maps each inheritance association to a foreign key relation from the special to the general class.
 4 Later, we discuss in which category the construction is carried out.

combination of the two variants, i.e., by a pair of morphisms: $M \xleftarrow{l} K \xrightarrow{r} N$. The pair (l, r) represents an arbitrary relation between M and N and can model:

1. Deletion of model objects, i.e., l is not surjective,
2. Addition of model objects, i.e., r is not surjective,
3. Renaming of model objects, i.e., l and r are bijective but not identities,
4. Splitting or unfolding of model objects, i.e., l is not injective, and
5. Gluing or folding of model objects, i.e., r is not injective.

Given a typed database $t: D \rightarrow M$ and a model refactoring $M \xleftarrow{l} K \xrightarrow{r} N$, we want to canonically construct the induced migration to some typed database $u: E \rightarrow N$. As a first step, we can use the pullback construction of t and l , which shall result in a typed database $v: F \rightarrow K$. For reasons of symmetry, we need to construct a pullback complement of v and r in the second step.

$$\begin{array}{ccccc}
 M & \xleftarrow{l} & K & \xrightarrow{r} & N \\
 \uparrow t & (1) & \uparrow v & (2) & \uparrow u \\
 D & \xleftarrow{l'} & F & \xrightarrow{r'} & E
 \end{array}$$

Unfortunately, such a pullback complement is not guaranteed to exist nor need be unique if it exists (see Fig. 2).

Even worse, there is no simple property for r that guarantees existence and uniqueness of the pullback complement. Some authors argue that r being epimorphism is sufficient, compare [16] or [17]. This is wrong as the following examples demonstrate.

Example 1 (Ambiguous Pullback Complements). Consider the situation depicted in Fig. 3 in the usual category of graphs and graph morphisms. The epimorphism f and the morphism g do not possess a unique (up to isomorphism) pullback complement, since (g, f_1^*) is pullback of (f, g_1^*) and (g, f_2^*) is pullback of (f, g_2^*) but D_1 and D_2 are not isomorphic. \square

In the category of sets and mappings, however, pullback complements seem to be uniquely determined. This is not (really) true, as is demonstrated by the following example.

Example 2 (Ambiguous Pullback Complements in Set). Let $g: \{1,2,3,4\} \rightarrow \{a,b\}$ be given by $g(1)=a$, $g(2)=a$, $g(3)=b$, $g(4)=b$ and $f: \{a,b\} \rightarrow \{3\}$ be the constant function as in Fig. 4. There are two pullback complements:

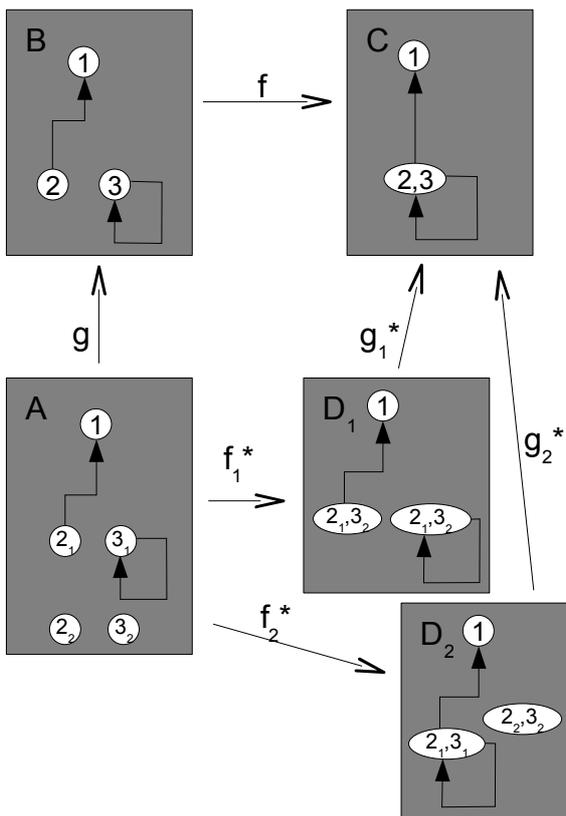


Fig. 3: Ambiguous Pullback Complement

There are two pullback complements:

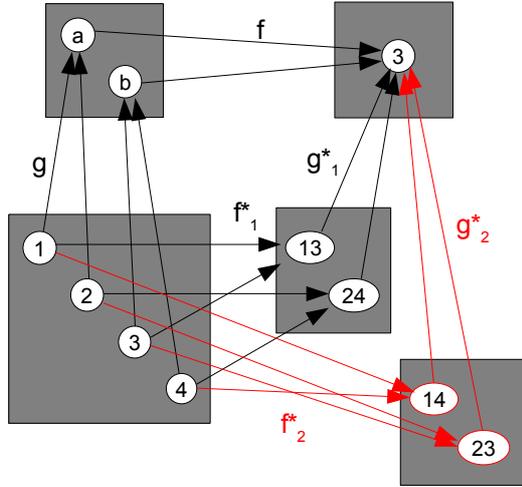


Fig. 4: Ambiguous pullback complement in *SET*

1. $(\{13,24\}, f^*_1: \{1,2,3,4\} \rightarrow \{13,24\}, g^*_1: \{13,24\} \rightarrow \{3\})$ with $f^*_1(1)=13, f^*_1(2)=24, f^*_1(3)=13,$ and $f^*_1(4)=24$.
2. $(\{14,23\}, f^*_2: \{1,2,3,4\} \rightarrow \{14,23\}, g^*_2: \{13,24\} \rightarrow \{3\})$ with $f^*_2(1)=14, f^*_2(2)=23, f^*_2(3)=23,$ and $f^*_2(4)=14$.

Obviously, $\{13,24\}$ and $\{14,23\}$ are isomorphic. But no isomorphism $i: \{13,24\} \rightarrow \{14,23\}$ translates f^*_1 to f^*_2 , in the sense: $i \circ f^*_1 = f^*_2$. Hence, we have isomorphic pullback complement objects. But the induced morphisms are ambiguous since they cannot be compared by the existing isomorphisms. \square

This type of ambiguity cannot be accepted in our context, since the morphisms represent the transition of the data from the old to the new model. There seems to be no chance to avoid this type of ambiguity, if we do not put additional requirements on the “vertical” morphisms g, g_1^* and g_2^* . These properties shall single out a unique choice for the pullback complement extension of g .

These examples provide the motivation for the following definitions:

Definition 3 (Graph). The category \mathbf{G} of *graphs* is the algebraic category w. r. t. the signature:

Sorts: $O(\text{bject})$

Ops: $s(\text{ource}), t(\text{arget}): O \rightarrow O$.

This is a simple form of graphs where we do not distinguish between nodes and edges. In such a graph, nodes can be characterized as objects n such that $s(n) = n = t(n)$. Graphs and graph morphisms of this type provide more flexibility in the refactoring/migration context we are considering here, for example: if two nodes x and y are mapped to the same node z , it is possible that a morphism maps an edge e with $s(e) = x$ and $t(e) = y$ to z , too. E.g. in Fig. 1, $l(\text{Unit}) = l(\text{Dept.}) = \text{Dept.}$ and the edge between them is mapped to the node Dept. as well.

Definition 4 (Component Graph). A *component graph* $g: G \rightarrow \underline{G}$ is a morphism in \mathbf{G} . A *component graph morphism* $\alpha: (g: G \rightarrow \underline{G}) \rightarrow (h: H \rightarrow \underline{H})$ is a pair $(\alpha: G \rightarrow H, \underline{\alpha}: \underline{G} \rightarrow \underline{H})$ such that the resulting square commutes, i.e., $\alpha \circ g = h \circ \underline{\alpha}$. The comma category \mathbf{CG} consists of all component graphs and all morphisms between them.

If not otherwise stated, we just write g for a component graph $g: G \rightarrow \underline{G}$. Note that G is the underlying graph and g provides a decomposition of G into parts or components via the the congruence $\text{kern}(g)^5$. Thus for the carrier set G we have $G = \Sigma \{[x]_g : x \in G\}$ where $[]_g$ denotes congruence classes of $\text{kern}(g)$. We also note that congruence classes are not necessarily subgraphs of G as can easily be seen in component graphs $id: G \rightarrow G$ where G contains edges.

⁵ The relation $\text{kern}(f)$ denotes the congruence that the morphism f induces on its domain, i.e., $(x, y) \in \text{kern}(f)$ iff $f(x) = f(y)$.



The additional component structure on graphs provides means to distinguish typings from refactorings. In a typing, we require that all components are instantiated completely in a 1:1 manner. In a refactoring we allow identification of objects if and only if they belong to the same component. Hence, refactorings map components injectively and typings map objects within components bijectively.

Note that **CG** has all limits and that pullbacks in **CG** can be constructed component-wise.

Definition 5 (Typings, Refactorings, and Migrations).

A *typing* $t: g \rightarrow h$ is a **CG**-morphism if for each $x \in G$ the mapping $t: [x]_g \rightarrow [t(x)]_h$ considered as a **SET**-morphism is bijective.

A *refactoring* is a pair of morphisms $m \xleftarrow{l} k \xrightarrow{r} n$ in **CG** such that l and r are injective. The morphisms l and r are called *refactoring morphisms* in this case.

A refactoring $m \xleftarrow{l} k \xrightarrow{r} n$ and a typing $t: d \rightarrow m$ induce a *migration* from typing $t: d \rightarrow m$ to typing $u: e \rightarrow n$, if there is a diagram as depicted to the right that satisfies:

$$\begin{array}{ccccc}
 m & \xleftarrow{l} & k & \xrightarrow{r} & n \\
 \uparrow t & & \uparrow (1) v & & \uparrow (2) u \\
 d & \xleftarrow{l'} & f & \xrightarrow{r'} & e
 \end{array}$$

1. (1) and (2) are pullbacks, and
2. r' is epimorphism.

The proof of the following proposition is straightforward and relies on the fact, that pullbacks preserve monomorphisms and isomorphisms.

Proposition 6 (Refactorings, Typings, and Pullbacks). If $(n^*: l \rightarrow g, m^*: l \rightarrow k)$ is the pullback of $(n: k \rightarrow h, m: g \rightarrow h)$ in **CG**, then

1. m^* is a refactoring if m is,
2. n^* is a typing if n is, and
3. if n is injective on components, i.e., $\forall x, y \in K: (n(x)=n(y) \wedge k(x)=k(y)) \Rightarrow x=y$, then the same property holds for n^* , i.e., $\forall x, y \in L: (n^*(x)=n^*(y) \wedge l(x)=l(y)) \Rightarrow x=y$

Proposition 7 (Existence and Uniqueness of Migrations). Let $m \xleftarrow{l} k \xrightarrow{r} n$ be a refactoring and let $t: d \rightarrow m$ be a typing. If $r: K \rightarrow N$ is an epimorphism, then:

1. There is a migration as defined in definition 5 and
2. The result of the migration is uniquely determined (up to isomorphism).

Proof. Subdiagram (1) can be constructed as a pullback. Thus (F, v, l') are unique up to isomorphism. The morphism v is a typing due to proposition 6. Having a typing v and a refactoring morphism r , we construct diagram (2), i.e., (E, r', u) , as follows and depicted in Fig. 5:

If the component graph f is the morphism $f: F \rightarrow E$, then

1. r' is the identity on \underline{E} ,
2. $E = F / \equiv$ where $\equiv = \text{kern}(f) \cap \text{kern}(r \circ v)$
3. $r' = [\]_{\equiv}$,
4. $\underline{u} = r' \circ v$,
5. u is the unique morphism providing $u \circ r' = r \circ v$ which exists since $\text{kern}(r \circ v) \supseteq \equiv$, and
6. component graph $e: E \rightarrow \underline{E}$ is the morphism with $e \circ r' = f$ which exists since $\text{kern}(f) \supseteq \equiv$.

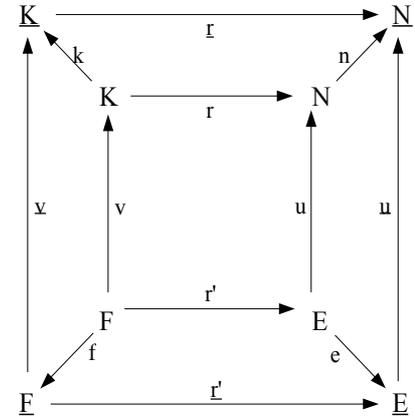


Fig. 5: Constructing the right side of a refactoring

By construction $u \circ r' = r \circ v$ and r' is epimorphism. Since $\text{kern}(r') = \text{kern}(r \circ v)$ on each component and r is an epimorphism, u is bijective on components and thus a typing. And it is easy to show that (v, r') is pullback of (r, u) in **SET** and therefore in **CG**: if there is o such that $r(x(o)) = u(y(o))$, then choose $o' = r'^{-1}(y(o)) \cap v^{-1}(x(o))$. This is unique, since v is bijective on components and, by construction, r' folds on components only. This completes the proof of the first statement.

To prove the second statement, let $(r^*: F \rightarrow E', u^*: E' \rightarrow N)$ be any other completion with the required properties. It is easy to see, that the two pullback situations project to pullback situations in **SET** on each component. Here we have $u^* \circ r^* = u \circ r'$ where u^* and u are bijective. Hence $\text{kern}(r^*) = \text{kern}(r')$ on each component of F . Because r is a refactoring, so are r' and r^* (see proposition 6) such that this property holds throughout F . Thus $E \cong E'$. \square

Although the framework presented so far allows copying and gluing of objects *within the same component* only, it provides some nice features for our purposes of information system refactoring, as the following example demonstrates.

Example 8 (Association Redirection). In Fig. 1 we showed how to introduce a superclass *Unit*. Subsequently, one needs to check the references to *Department*-objects and redirect them to *Unit*-objects if necessary. To do this, consider the model refactoring in Fig. 6⁶: All three graphs have 3 components; the non-trivial component in each graph (the component that has more than one element) is highlighted. Using this refactoring in a migration redirects all associations of type “7” from the source of “6” to the target of “6”. It uses an intermediate vertex “2”, that is introduced by the left-hand side l as an unfolding and removed again by the right-hand side r

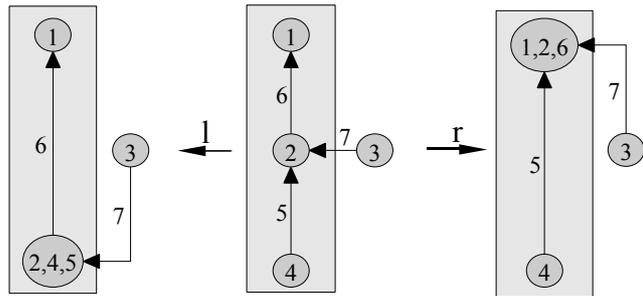


Fig. 6: Redirection of associations

⁶ We indicate the model objects numerically to clarify the mappings.

by a corresponding folding. This example shows, that we are able to redirect association sources and targets as long as we stay in the same component. \square

With these features, we should be able to handle all refactorings that are concerned with inheritance structures. Recall, that inheritance can be considered as some sort of static composition between objects: an object of class c can be considered to be composed of a set of (sub-)objects, namely one object for each direct or indirect ancestor class c' of c . All these objects are created at the same time the most special object is created. And they are also destroyed at the same time. Hence, we can model them as explicit parts in a component graph on the instance level in our framework.

But these components are not components in the sense of typings (Def. 5). It is not the case, that the *complete* inheritance tree of classes needs to be instantiated, if one class is. If there are concrete classes that possess subclasses, an object might instantiate a proper subpart of the complete inheritance tree of its class, only. Our approach is not able to handle those incomplete parts, since pullback complements do not always exist in these situations.

Example 9 (Missing Pullback Complement). Consider the reverse process as in example 8. An association to class “1,2,6” (a concrete superclass of “4”) shall be redirected to its subclass, see Fig. 7. We apply this rule to an instance of “1,2,6”, called “1,2,6'”. The pullback on the left produces an intermediate object “2'” in F. But we can easily deduce, that the right part is not able to complete the diagram to a double-pullback situation. This is mainly due to the fact that the non-trivial component in K is only partially instantiated in F (there is no “4”-object). For suppose, that such a pullback complement $r': F \rightarrow E$, $u: E \rightarrow N$ exists. Then $r'(2')$ is a preimage of “2,4,5” under u . This is only possible if there is a preimage of “4” in F. \square

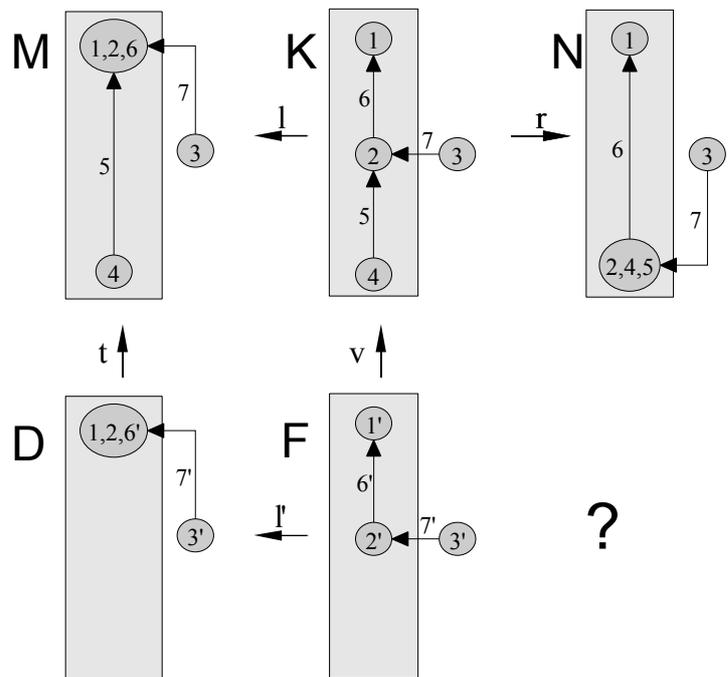


Fig. 7: Missing pullback complement

We might use a trick to handle inheritance. We always instantiate complete inheritance graphs, when an object is created and keep the information about the most special *real* object in the resulting part (of *real* and *extra* objects). Then we distinguish two views on the system: (1) the *refactoring perspective* and (2) the *operational perspective*. In the first perspective, all objects are visible and our framework is applicable. The second perspective blends out all extra objects in order to keep the system's state consistent from



the operational point of view.⁷ With these additional arrangements, inheritance structures and the typical refactorings could be modeled.

But there are also disadvantages of this approach: the additional instantiations might cause a significant memory overhead. In this paper, we use a different approach, which omits this problem: In the next section, we slightly generalize our framework such that partial instantiations of components in the model are allowed. To achieve that, we do no longer require that the right-hand side of a migration is a pullback.

3. Partial Instantiation of Components

In this section, we relax the requirements for typings. *Weak typings* allow partial instantiations of model components, since they are injective on each component but need not be surjective.

Definition 10 (Weak Typing). A component graph morphism $\alpha: (g: G \rightarrow \underline{G}) \rightarrow (h: H \rightarrow \underline{H})$ is a *weak typing* if it is injective on each component, i.e., $\alpha(x) = \alpha(y) \wedge g(x) = g(y) \Rightarrow x = y$.

Now we use the construction in the proof of proposition 7 to construct the right-hand side of a migration. This works for weak typings as well.

Construction 11 (Folding). Consider Fig. 8, where weak typing n and refactoring morphism m are given. We construct the *folding completion* of this situation as follows:

1. $m^* = ([\]_{\equiv}, \text{id}_{\underline{G}})$, where $\equiv = \text{kern}(g) \cap \text{kern}(m \circ n)$,
2. $n^* = (i, \underline{m} \circ \underline{n})$, where i is the unique morphism with $i \circ [\]_{\equiv} = m \circ n$, since $\text{kern}(m \circ n) \supseteq \equiv$, and
3. the component graph j is the unique morphism with $j \circ [\]_{\equiv} = \text{id} \circ g$, since $\text{kern}(g) \supseteq \equiv$.

m^* is a CG-morphism by construction. Moreover, we obtain $k \circ i = m \circ n \circ j$, since $[\]_{\equiv}$ is epi and $k \circ i \circ [\]_{\equiv} = k \circ m \circ n = \underline{m} \circ \underline{n} \circ g = \underline{m} \circ \underline{n} \circ j \circ [\]_{\equiv}$. Thus, n^* is a CG-morphism, too. □

Lemma 12 (Folding). If (m^*, n^*) is folding of (m, n) , m^* is refactoring morphism and n^* is typing.

Proof. The first part is obvious, since \underline{m}^* has been constructed as the identity which is a mono.

For the proof of the second statement let $i(x) = i(y)$ and $j(x) = j(y)$. Now consider arbitrary preimages x' and y' for x and y wrt. $[\]_{\equiv}$, i.e., $[x']_{\equiv} = x$ and $[y']_{\equiv} = y$. Since $j \circ [\]_{\equiv} = \text{id} \circ g$, we conclude $g(x') = g(y')$. Since $i \circ [\]_{\equiv} = m \circ n$, it follows that $m(n(x')) = m(n(y'))$.

Thus, $(x', y') \in \text{kern}(g) \cap \text{kern}(m \circ n)$, which means that $[x']_{\equiv} = [y']_{\equiv}$. Hence, $x = y$. □

⁷ Note that the model is stable under the operational perspective!

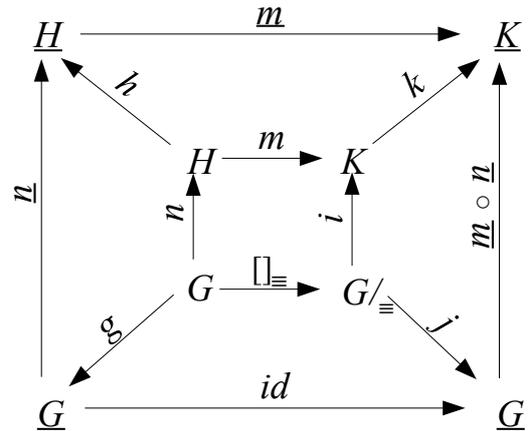


Fig. 8: Construction of a Folding



Folding diagrams possess an interesting universal property as the following proposition shows.

Proposition 13 (Initiality of Foldings). Let the pair of morphisms $(m^*: g \rightarrow f, n^*: f \rightarrow k)$ be the folding of a weak typing $n: g \rightarrow h$ and a refactoring morphism $m: h \rightarrow k$ as it is constructed in construction 11. Then for every triple of morphisms $(w: g \rightarrow b, t: b \rightarrow a, v: k \rightarrow a)$ such that t is weak typing and $t \circ w = v \circ m \circ n$, there is a unique morphism $u: f \rightarrow b$ with $t \circ u = v \circ n^*$ and $u \circ m^* = w$.

Proof. Let the folding be given as in Fig. 8. We set $\underline{u} = \underline{w}$ and get immediately (1) $\underline{u} \circ m^* = \underline{u} \circ \text{id} = \underline{u} = \underline{w}$. We show that $\underline{u} \in \text{kern}(w)$. Let $m(n(x)) = m(n(y))$ and $g(x) = g(y)$. It follows $t(w(x)) = v(m(n(x))) = v(m(n(y))) = t(w(y))$ and $b(w(x)) = \underline{w}(g(x)) = \underline{w}(g(y)) = b(w(y))$. Since t is weak typing, we get $w(x) = w(y)$ as desired. Now there is a unique $u: G/\equiv \rightarrow B$ with (2) $u \circ [\]_{\equiv} = \underline{u} \circ m^* = \underline{w}$. Since $b \circ u \circ [\]_{\equiv} = b \circ w = \underline{w} \circ g = \underline{w} \circ j \circ [\]_{\equiv} = u \circ j \circ [\]_{\equiv}$, we can conclude (3) $b \circ u = \underline{u} \circ j$. And $t \circ u \circ [\]_{\equiv} = t \circ w = v \circ m \circ n = v \circ i \circ [\]_{\equiv}$ provides (4) $t \circ u = v \circ i = v \circ n^*$. Finally, we also have (5) $t \circ \underline{u} = v \circ m \circ n = v \circ n^*$. \square

Proposition 13 characterizes foldings up to isomorphism. In the following, we say that a diagram is an abstract folding if it has the property of proposition 13:

Definition 14 (Abstract Folding). As depicted in Fig. 9, a pair $(m^*: g \rightarrow f, n^*: f \rightarrow k)$ consisting of a refactoring morphism m^* and a weak typing n^* is the *abstract folding* of a weak typing $n: g \rightarrow h$ and a refactoring morphism $m: h \rightarrow k$ if (1) $n^* \circ m^* = m \circ n$ and (2) for every triple $(w: g \rightarrow b, t: b \rightarrow a, v: k \rightarrow a)$ such that t is weak typing and $t \circ w = v \circ m \circ n$, there is a unique morphism $u: f \rightarrow b$ with $t \circ u = v \circ n^*$ and $u \circ m^* = w$.

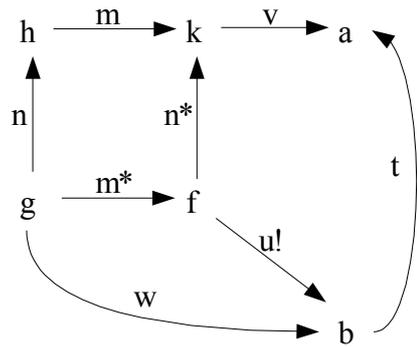


Fig. 9: Abstract Folding

Corollary 15 (Uniqueness of Abstract Foldings). Two abstract foldings of a weak typing $n: g \rightarrow h$ and a refactoring morphism $m: h \rightarrow k$ coincide up to isomorphism. Hence $(m^*: g \rightarrow f, n^*: f \rightarrow k)$ is the abstract folding if and only if the statement

$$m^* \text{ is epimorphism and } m^*(x) = m^*(y) \Leftrightarrow (g(x) = g(y) \wedge m(n(x)) = m(n(y)))$$

holds.

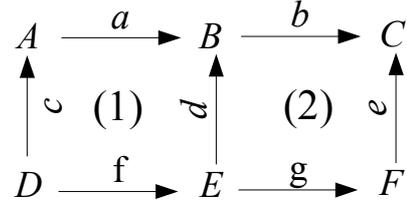
Proof. Direct consequence of Definition 14 and the fact that the first folding compares to the second and vice versa. Therefore, we get two morphisms between the two foldings, which must be inverse morphisms, because their composition coincide with the identity on the folding objects (unique morphism from a folding to itself). \square

Abstract foldings enjoy the same composition and decomposition properties as pushouts or pullbacks.


Proposition 16 (Composition and Decomposition of Abstract Foldings).

Consider the situation depicted in the diagram below⁸.

1. If the squares (1) and (2) are abstract foldings, then the rectangle (1) + (2) is an abstract folding.⁹
2. If the rectangle (1) + (2) and the square (1) are abstract foldings, then (2) is an abstract folding.
3. If typing e and refactoring h is the abstract folding of typing c and refactoring $b \circ a$, it can be decomposed into two foldings as in the diagram on the right, where $h = g \circ f$, if the underlying category has all abstract foldings.


Proof.

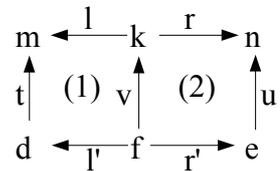
(1) Let morphisms v, t, w , be given such that t is typing and $t \circ w = v \circ b \circ a \circ c$. Since (1) is abstract folding, we get u_1 such that $u_1 \circ f = w$ and $t \circ u_1 = v \circ b \circ d$. Now u_1, t and v compare to (2) and we get u_2 with $u_2 \circ g = u_1$ and $t \circ u_2 = v \circ e$. Substituting $u_2 \circ g = u_1$ in $u_1 \circ f = w$ provides $u_2 \circ g \circ f = w$. For the proof of uniqueness, let morphism u_3 be given such that $u_3 \circ g \circ f = w$ and $t \circ u_3 = v \circ e$. Then $u_3 \circ g \circ f = u_2 \circ g \circ f$ and $t \circ u_3 \circ g = v \circ b \circ d = t \circ u_2 \circ g$ hold. We obtain $u_3 \circ g = u_2 \circ g$, since (1) is abstract folding. But this implies $u_3 = u_2$, since (2) is abstract folding.

(2) Let v, t, w , be given such that t is typing and $t \circ w = v \circ b \circ d$. It follows $t \circ w \circ f = v \circ b \circ a \circ c$. Since (1)+(2) is abstract folding, there is u such that $u \circ g \circ f = w \circ f$ and $t \circ u = v \circ e$. We also have $t \circ u \circ g = v \circ e \circ g = v \circ b \circ d$. Since (1) is abstract folding, we get $u \circ g = w$. Uniqueness follows from the uniqueness of u for (1)+(2).

(3) If there are all abstract foldings, we can construct (d, f) as a folding, which provides diagram (1). The morphism g is obtained as the unique completion of the diagram from the folding (1). That diagram (2) is an abstract folding follows from (2) of this proposition. \square

Definition 17 (Generalized Migration). A refactoring $m \xleftarrow{l} k \xrightarrow{r} n$ and a weak typing $t: d \rightarrow m$ induce a *generalized migration* from $t: d \rightarrow m$ to weak typing $u: e \rightarrow n$, if there is a diagram as depicted to the right that satisfies:

1. Subdiagram (1) is pullback and
2. Subdiagram (2) is abstract folding.


Theorem 18 (Existence and Uniqueness of Generalized Migrations).

Given a weak typing $t: D \rightarrow M$ and refactoring $M \xleftarrow{l} K \xrightarrow{r} N$ for the model of t , there is an induced migration and the result of the migration is unique up to isomorphism.

⁸ Here, for the sake of readability, CG-objects are presented in capital letters.

⁹ (1)+(2) consists of the morphisms $b \circ a, e, g \circ f$, and c .

Proof. Direct consequence of (1) the existence and uniqueness of pullbacks in **CG**, (2) the fact that pullbacks in **CG** preserve weak typings (proposition 6, 3.), and (3) the existence and uniqueness of abstract foldings in **CG** (Construction 11 and Corollary 15). \square

Theorem 18 justifies that we write $R(t)$ for the result typing of a migration from a typing $t: D \rightarrow M$ using a refactoring $R = M \xleftarrow{l} K \xrightarrow{r} N$. Fig. 10 shows the generalized migration that we searched for in Fig. 7.

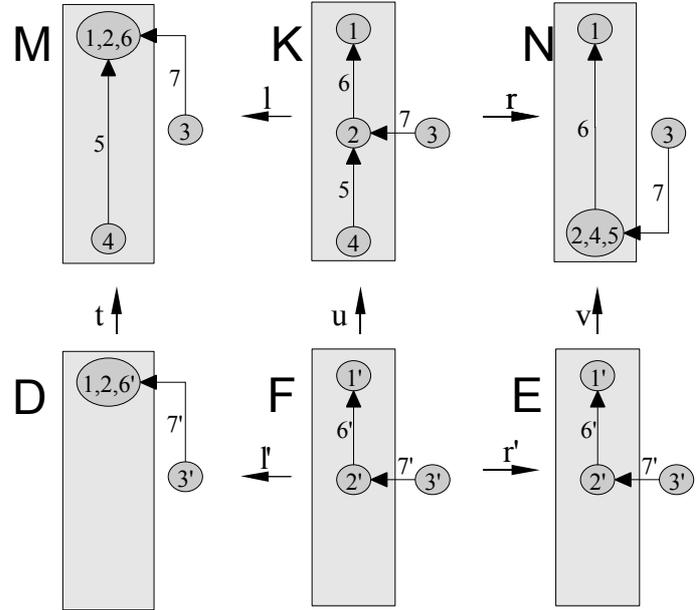


Fig. 10: A generalized migration

4. Sequential Composition

In this section, we show that there is a natural sequential composition $R_2 \circ R_1$ of refactorings R_1 and R_2 and that applying a sequential composition to a weak typing t provides exactly the same result as the sequence of first applying R_1 to t and second R_2 to $R_1(t)$, i.e., $R_2 \circ R_1(t) = R_2(R_1(t))$.

Definition 19 (Sequential Composition of refactorings).

The *sequential composition* $R_2 \circ R_1 = (l_1 \circ p_1: J \rightarrow M, r_2 \circ p_2: J \rightarrow P)$ of two refactorings $R_1 = (l_1: K \rightarrow M, r_1: K \rightarrow N)$ and $R_2 = (l_2: H \rightarrow N, r_2: H \rightarrow P)$ is defined with the help of the pullback object $(p_1: J \rightarrow K, p_2: J \rightarrow H)$ of r_1 and l_2 as depicted in Fig. 11.

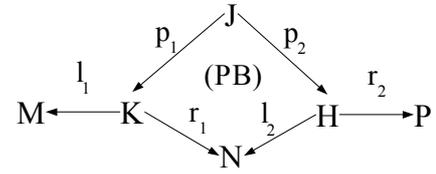


Fig. 11: Sequential composition

Note that the sequential composition is well-defined due to proposition 6, 1. and the fact that the composition of two refactoring morphisms is a refactoring morphism again.

In order to prove our main theorem, i.e., $R_2 \circ R_1(t) = R_2(R_1(t))$, we need the following technical lemma.

Lemma 20 (Pullback Cubes Preserve Abstract Foldings). Consider the commuting diagram in **CG** below¹⁰. If the pair of morphisms (i, q) is the abstract folding of the morphism pair (r, m) , the pair (p, t) is the pullback of the pair (s, q) , and (j, v) is the pullback of (t, i) , then the pair of morphisms (j, p) is the abstract folding of (n, k) .

¹⁰ Here, we depict **CG**-objects as arrows with filled tip.

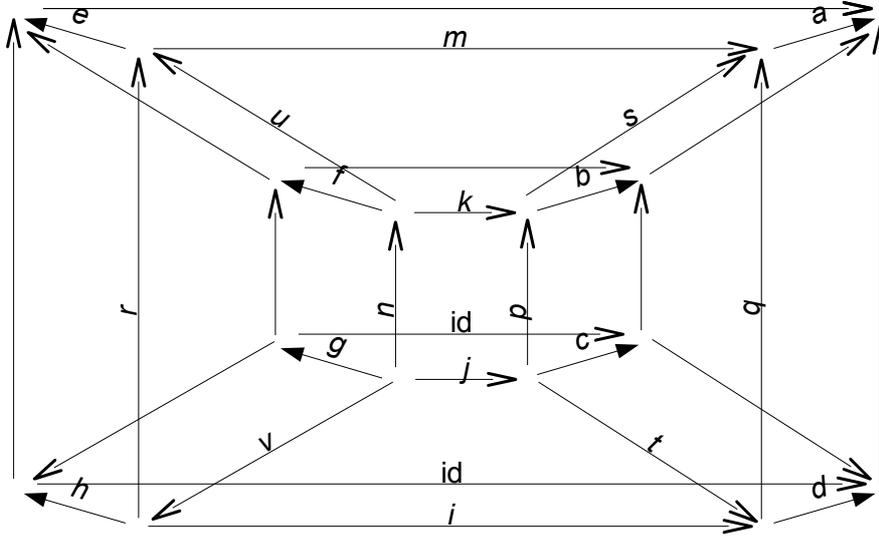


Fig. 12: Pullback Cube

Proof. The assumptions of the lemma provide that i is the identity. Since pullbacks preserve isomorphisms, we can set $j = \text{id}$ without loss of generality. Because the bottom face is a pullback and i is an epimorphism (see Corollary 15), j is an epimorphism as well. Again, from Corollary 15 we deduce that it suffices to show, that

$$j(x) = j(y) \Leftrightarrow [k(n(x)) = k(n(y)) \wedge g(x) = g(y)]$$

holds for all $x, y \in G$.

“ \Rightarrow ”: (1) $j(x) = j(y) \Rightarrow p(j(x)) = p(j(y)) \Rightarrow k(n(x)) = k(n(y))$

(2) $j(x) = j(y) \Rightarrow c(j(x)) = c(j(y)) \Rightarrow \text{id}(g(x)) = \text{id}(g(y)) \Rightarrow g(x) = g(y)$

“ \Leftarrow ”: Let $k(n(x)) = k(n(y)) \wedge g(x) = g(y)$ be given. Since (p, t) is pullback, it is sufficient to show: (3) $t(j(x)) = t(j(y)) \wedge$ (4) $p(j(x)) = p(j(y))$:

(3) (a) $g(x) = g(y) \Rightarrow v(g(x)) = v(g(y)) \Rightarrow h(v(x)) = h(v(y))$.

(b) $k(n(x)) = k(n(y)) \Rightarrow s(k(n(x))) = s(k(n(y))) \Rightarrow m(r(v(x))) = m(r(v(y)))$.

Since (q, i) is abstract folding, it follows from Corollary 15, (a) and (b) that $i(v(x)) = i(v(y))$, which provides $t(j(x)) = t(j(y))$, because $t \circ j = i \circ v$.

(4) $k(n(x)) = k(n(y)) \Rightarrow p(j(x)) = p(j(y))$. □

Theorem 21 (Sequential Composition). If $R_2(R_1(t))$ for two refactorings R_1 and R_2 is defined, we have $R_2 \circ R_1(t) = R_2(R_1(t))$.

Proof. Consider the following diagram, which depicts $R_2(R_1(t))$. This migration sequence is given by the four squares (1) MKFD, (2) KNEF, (3) NHCE, and (4) HPBC. (1) and (3) are pullbacks and (2) and (4) are abstract foldings. The additional material in the diagram is defined as follows: The pair of morphisms (p_1, p_2) is given as a pullback of r_1 and l_2 , compare construction of $R_2 \circ R_1$ in definition 19. We write (5) for the resulting square NKJH. We construct (p'_1, p'_2) as pullback of (r'_1, l'_2) . We write (6) for the resulting square EFIC. The morphism u_2 is the universal completion of the diagram into the pullback object J . Now, the square (7) KJIF is pullback as well. This is due to the fact that (3)+(6) is pullback¹¹, (3)+(6) = (5)+(7), and (5) is pullback¹². The square (8) JICH is abstract folding due to Lemma 20. Now diagram (1)+(7) is pullback, since pullbacks compose. It is the left-

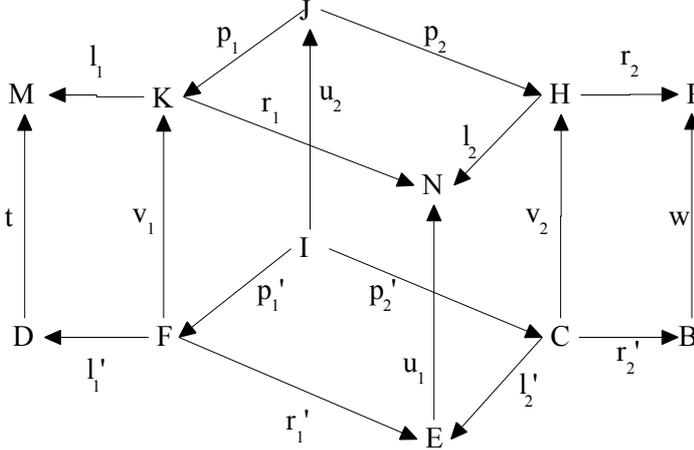


Fig. 13: Migration sequence

hand side of the migration induced by $R_2 \circ R_1$. Diagram (8)+(4) is abstract folding, since abstract foldings compose (compare Proposition 16, 1.). It is the right-hand side of the migration induced by $R_2 \circ R_1$. This together shows that $R_2 \circ R_1$ migrates t to w as well. \square

With Theorem 21 we are, on the one hand, able to compose long refactoring sequences into one single refactoring, which can capture the effect of the whole sequence. On the other hand, we can decompose complex refactorings into a composition of simpler ones.

5. General Framework

The whole approach presented above is almost independent from the underlying category of graphs resp. component graphs. What we need for the existence and uniqueness of migrations is the existence of pullbacks and abstract foldings. For the results concerning sequential composition, we need the cube lemma 20, i.e., that pullbacks “pull back” abstract foldings. Thus, we can present our requirements for a category to provide the infrastructure for unique migrations and sequential compositions as follows:

An abstract *migration framework* is a category C together with two subcategories T and R which have the same objects as C . The morphisms in T are called *typings* and the morphisms in R are called *refactoring morphisms*. The system (C, T, R) is subject to the following requirements:

- (1) C has all pullbacks
- (2) C has abstract foldings for all pairs of morphisms $(f: A \rightarrow B \in T, g: B \rightarrow C \in R)$.

¹¹ Composition property of pullbacks.

¹² Decomposition property of pullbacks.



- (3) Pullbacks in C preserve morphisms of T and of R .
- (4) In each cube with corners $K, N, H, J, F, E, C,$ and I , as it is depicted in Fig. 13, the square $JICH$ is an abstract folding if $KNEF$ is abstract folding and the squares $IFEC$ and $NECH$ are pullbacks.

Since abstract foldings are a generalization of surjective pullback complements, the framework presented in section 2 fits into this setting as well. Another instance is given by simple graphs, arbitrary morphisms as typings and injective morphisms as refactoring morphisms. Here we can use surjective pullback complements as abstract foldings as well [15].

6. Conclusion

We propose formalizations of aspects in the process of refactoring information systems. The power of our attempt is that a model refactoring can uniquely and automatically be extended to the instance level. In contrast to other more practical solutions, we can prove correctness of our approach. The framework is described using abstract notions from category theory.

With a strong assumption to the typing morphisms we can generalize a migration to a double-pullback diagram. As a first step, it is possible to handle addition, renaming, and removal of model objects. The investigation under which conditions folding and unfolding is possible, leads to a model structure where one had to restrict to 1:1 associations on certain components. A refactoring morphism may fold or unfold on these components, only. In a second step we showed that these settings are correct as well.

However, object trees of inheritance structures are, in general, not completely instantiated. To treat this case in a similar way, we have to weaken the assumptions on the type mappings. But weak typings do not always lead to double-pullback constructions. Thus, this third step requires a generalization of pullback complements. We introduced abstract foldings that enjoy some of the well-known properties of pullbacks and pushouts. Abstract foldings are initial in a reasonable context, which reveals a uniqueness statement of generalized migrations and prepares a statement on the composition of refactorings.

Composing migrations into larger projects and decomposing migrations into smaller steps leads to the question if there is a minimal set of *atomic* refactorings, from which each refactoring can be constructed by sequential composition. This might be an interesting topic for future research as well as the question, under which conditions refactorings are parallel or sequential independent and can be performed concurrently. These results are valuable for tools that produce migrations on the basis of the construction of pullbacks and abstract foldings.

Finally, in a forth step, we describe a way of integrating refactoring and migration procedures in a more general framework that abstracts away from the underlying category. We define requirements that are the basis for a generalized system. These requirements are very similar to the axioms for adhesive categories in [14]. It is up to future research to investigate if both frameworks can be seen as two instances of an even more general system.



References

- 1 Havey, M.: Essential Business Process Modeling. O'Reilly 2005
- 2 Martin, R. C.: Agile Software Development, Principles, Patterns, and Practices. Prentice Hall 2002
- 3 Beck, K.: Extreme Programming Explained. Addison Wesley 2000
- 4 Beck, K.: Test-driven Development by Example. Addison-Wesley 2002
- 5 Fowler, M.: Refactoring: Improving the Design of Existing Code . Addison-Wesley 1999
- 6 Kerievsky, J. : Refactoring to Patterns. Addison-Wesley 2004
- 7 D'Anjou, J et al: The Java Developer's Guide to Eclipse. Addison-Wesley 2005
- 8 Ambler, S. W.: Agile Database Techniques. Wiley 2003
- 9 Ambler, S. W.: Refactoring Databases : Evolutionary Database Design . Addison-Wesley 2006
- 10 Hainaut, J.-L.: Introduction to database reverse engineering. LIBD Publish. 2002
- 11 Bauer, Ch., King, G. : Hibernate in Action . Manning Publications 2004
- 12 Löwe, M.: Evolution Patterns – A Graphical Framework for Software Redesign. Proceedings ISAS'99 1999
- 13 Adamek, J., Herrlich, H., Strecker G. E.: Abstract and Concrete Categories - the Joy of Cats. 2004 <http://katmat.math.uni-bremen.de/acc/acc.pdf>
- 14 Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer 2006
- 15 Löwe, M., König, H., Peters, M., Schulz, Ch.: A Formal Framework for Information System Refactorization. Proceedings WMSCI 2006, Vol. 1, 75-80, 2006
- 16 Meisen, J.: Pullbacks in Regular Categories. Canad. Math. Bull. Vol.16(2) 1973
- 17 Bauderon, M., Jacquet, H.: Pullback as a generic graph rewriting mechanism. Applied Categorical Structures Vol.9(1) 2001



An Approach to Invariant-based Program Refactoring

Tiago Massoni * and Rohit Gheyi * and Paulo Borba *

*Informatics Center — Federal University of Pernambuco
PO Box 7851 – 50.732-970 Recife, PE
{tlm,rg,phmb}@cin.ufpe.br

***Abstract.** Refactoring tools include checking of an object-oriented program for the fulfillment of preconditions, for ensuring correctness. However, program invariants – semantic information about classes and fields assumed valid during program execution – are not considered by this precondition checking. As a result, applicability of automated refactorings is constrained in these cases, as refactorings that would be applicable considering the invariants get rejected, usually requiring manual changes. In this paper, we describe initial work on the use of program invariants (declared as code annotations) to increase applicability of automated refactoring. We propose an approach that uses primitive program transformations that employ the invariant to make the program syntactically amenable to the desired refactoring, before applying the refactoring itself.*

1 Introduction

A popular technique for dealing with evolution-related problems is *refactoring* [Fow99, MT04], which improves software structure while preserving behavior to better support adaptations or additions. The practice of refactoring has been improved by supporting tools, avoiding manual work and increasing trust on behavior preservation. Usually a catalog of refactorings is offered, from which users can choose the desired transformation for the problem in context (for instance, push down fields of a class to a subclass, or introduction and renaming of classes and fields). These automated refactorings present *preconditions* that are checked against the code subject to refactoring, in order to ensure correctness.

Such precondition checking involves analysis of static information from the program – declarations and statements – enforcing strong conditions that limit the applicability of refactorings. While being effective to ensure safe refactorings, it leads to prevention of refactoring on programs that would be eligible if some semantic assumptions about the program behavior were taken into consideration. In these cases, extra manual refactoring is required, minimizing the benefits of using refactoring tools. Examples of such refactorings are presented in Section 2.

Semantic assumptions about the program can be expressed as *program invariants*, which consist in



predicates about the state of objects and their inter-relationships, assumed valid throughout every possible program execution. In this paper, we describe how invariants can be used to increase the applicability of some automated refactorings (*invariant-based refactoring*). We indicate that if certain types of invariants are assumed, transformations based on these invariants can be applied to programs that would not be eligible for refactoring using the current tools. Programmers may provide invariants on classes and their fields as *code annotations*, or invariants may be discovered by existing analysis tools.

We propose an approach for using a sequence of *primitive program transformations* that employ invariants for making the program syntactically amenable to the desired refactoring, before applying the refactoring itself. As the program initially does not satisfy the preconditions, we use the invariants as a basis for applying some auxiliary program transformations that results in a program that can be subject to the desired automated refactoring. As a consequence, programs not fulfilling preconditions may be automatically refactored based on information about their behavior. By that, refactoring tools are applicable to more programs. Besides using code annotations, this approach can be applied in combination with program verification tools for discovering invariants from programs, using static or dynamic analysis.

2 Problem Statement

In this section, we make the case for invariant-based refactoring, by describing examples of refactoring tool limitations related to their precondition checking. Figure 1 shows a partial Java program representing a file system. The superclass `FObject` defines a general representation for file system objects – files and directories – being specialized in correspondent subclasses. The `parent` field, declared into `FObject`, defines the parent directory for each directory object.

```

class FObject {
    Name name;
    Dir parent; ...
    Dir getParent(){
        return this.parent;
    }
}

class File extends FObject{
    int size;
    java.util.Date creationDate; ...
    File(){
        this.parent = null;
    }
}

class Dir extends FObject{
    List ownedFiles = new ArrayList();
    int numberOfEntries; ...
    Dir(Dir parent) {
        parent = parent;
        numberOfEntries = 0;
    } ...
}

```

Figure 1: File system implementation.



Suppose that it is known that the `parent` field is always null for non-directory objects, such as files. In this context, it is desirable to apply a refactoring for *pushing down* `parent` to the `Dir` class [Fow99]. Refactoring tools, such as Eclipse, offer this refactoring in their catalog; selecting `parent`, the tool performs all related changes to apply this refactoring.

However, some of the previous accesses to the field do not allow the application of the refactoring. As indicated in the highlighted statements of Figure 1, accesses within `FSObject` and `File` would be invalid after moving the field to `Dir`. Nevertheless, `parent` does not make sense to non-directory objects – for example, `File` only allows null assignments to the inherited field –, and this semantic information is not taken into account by refactoring tools. This refactoring cannot be applied correctly in existing tools; otherwise, the resulting program would present typing errors.

3 Program Invariants

Reasoning about design of object-oriented programs usually rely on a number of *object invariants*, which represent consistency conditions on the program's objects and its data fields that must be maintained throughout the execution of the program [LM04]. This information may be stated in separate artifacts, such as object models [LG01], or integrated into programming languages, using *annotation languages*. Code annotations can handle the complexities of object-oriented code, also being directly compilable into runtime assertions.

Examples of annotation languages for Java include the Java Modeling Language (JML) [BCC⁺05] and Alloy Annotation Language (AAL) [KMJ02]. We illustrate the use of annotations by showing object invariants for `FSObject` and `Dir`, in Figure 2. The invariants are based on simple first-order logic based on AAL. In `FSObject`, the invariant states that file system objects, except directories, have no parent directories; `FSObject - Dir` yields all `FSObject` instances that are not `Dir` instances. The set of dereferences of the `parent` attribute from these instances is always empty (# denotes set cardinality).

```
class FSObject {
  //@ invariant {
  //@ #((FSObject - Dir).parent) = 0
  //@ }
  Name name;
  Dir parent; ...
}
```

Figure 2: Object invariants as code annotations.

Code annotations are provided in at least three different ways. Users may add annotations as supplementary design information that may help program documentation and analysis. For instance, advanced static analysis can be applied to programs annotated with invariants [FLL⁺02]. Similarly, invariants may be transferred to the program by abstract models *in conformance*. For instance, the invariant from



Figure 2 could have been defined in a structural model (UML class diagrams with constraints) which the program conforms to. Conformance implies that if the model constrains the `parent` relationship to directories only, the classes implementing the involved concepts must follow this constraint.

Furthermore, likely invariants may also be discovered from several executions of a program, as seen with Daikon tool support [ECGN01]. It consists in a program analysis that generalizes over observed values to assume program properties, used in testing, verification and bug detection. In this case, user intervention for dealing with invariants is minimized.

4 Approach for Invariant-based Refactoring

In Section 2, we exemplified a program that would be eligible to automated refactorings if semantic information was considered. In this section, we show how program invariants that represent this semantic information can be used as a basis for increasing the applicability of such automated refactorings. We first show a systematic method for applying behavior-preserving transformations to programs (Section 4.1), that will aid making programs subject to refactoring. These transformations build a foundation for our approach, as described in Section 4.2. Other examples of invariants that could be applied accordingly are defined in Section 4.3.

4.1 Primitive Program Transformations

Refactoring must preserve the observable behavior of a program. A way to facilitate mechanization of refactorings in tool support is to adopt an *algebraic* method, in which a refactoring is made of a sequence of behavior-preserving transformations [Opd92]. One classical approach for defining these transformations can be *primitive* laws [HHJ⁺87] relating language constructs. Easily mechanized by term rewriting, these laws are immediately available as a framework for transforming programs. In addition, primitive laws may be composed for deriving large-grained transformations that preserve semantics of programs. For object-oriented programming, an extensive set of laws has been defined for the Refinement Object-Oriented language (ROOL) [B⁺04], which is correspondent to a subset of Java.

Although defined for a simplified language, most laws can be leveraged to Java. The most critical restriction is on its *copy semantics*, rather than a reference semantics [B⁺04]. In ROOL, objects are treated as primitive values (records), consequently presenting no pointers. The laws showed in this section do not deal with object sharing; however, for laws that depend on sharing to be correct, an additional property of *confinement* must be ensured. In fact, the laws must be revised in order to deal with reference semantics. In this section, the laws are presented following the Java syntax, for simplicity.

As an example of primitive law, the following moves an field up (applying from left to right) or down (right to left) to super or subclass, respectively, which is put in practice by refactoring tools. Each law denotes two transformations, as it defines equivalence. The provisos (preconditions) ensure that the transformations denoted by the law preserve semantics. The equivalence is valid within a context of class declarations *cds* and a main method *c*. The symbol ‘(→)’ before the first proviso indicates it is only required for applications of this law from left to right. On the other hand, ‘(←)’ is used when a proviso is necessary only for applying a law from right to left.

**Law 1** *(move field to superclass)*

<pre>class B extends A { ads ops } class C extends B { a : T; ads' ops' }</pre>	$=_{cds,c}$	<pre>class B extends A { a : T; ads ops } class C extends B { ads' ops' }</pre>
---	-------------	---

provided

- (\rightarrow) The field name a is not declared by the subclasses of B in cds ;
- (\leftarrow) $D.a$, for any $D \leq B$ and $D \not\leq C$, does not appear in cds , c , ops , or ops' .

The ads identifier represents field declarations in the class, while ops stands for the declaration of methods and constructors. The notation $B.a$ denotes uses of a through expressions whose static type is exactly B (for instance, an expression yielding an object from class B , strictly). To denote that B is a subclass of A , we write $B \leq A$. The second proviso above precludes an expression such as `this.a` from appearing in ops , but does not preclude `this.c.a`, for an field $c : C$ declared in B . The last expression is valid in ops no matter whether a is declared in B or in C .

Several laws for commands and expressions complement laws for object-oriented constructs. For instance, the next law allows us to introduce type casts to expressions, as long as the type of the expression is consistent with the cast, given the type context (denoted by \triangleright).

Law 2 *(introduce trivial cast in expressions)*

If $cds, A \triangleright e : C$, then $cds, A \triangleright e = (C)e$.

For ensuring that the program's state before a given statement fulfills a given invariant, we make use of Java assertions. Each assertion contains a boolean expression that you believe will be true when the assertion executes. If it is not true, the system will throw an error. We can, for example, extract assertions from guards, as stated by the following law, where ψ_i denotes a boolean formula.

Law 3 *(assertion condition)*

if (ψ_i) **{** c_i **}** = **if** (ψ_i) **{** *assert* (ψ_i) ; c_i **}**

4.2 An Approach for Invariant-based Refactoring

We now describe an approach for applying behavior-preserving transformations for refactoring programs based on invariants. A type of invariant is identified; for that invariant, the tool can apply a predefined sequence of primitive laws of programming which we call *strategy*. Strategies possess two key properties: (1) they must rewrite programs for updating statements, using invariants, before the desired refactoring; (2) they preserve program behavior, which entails from the application of laws.

Checking the program from Figure 1 in current refactoring tools involves type declarations for ensuring correctness. As the parent field is to be pushed down to a particular subclass, accessing parent



from a reference whose type is not of that subclass would be invalid. On the other hand, an intuitive analysis of the program in the light of the invariant from Figure 2 shows some interesting aspects about those fragments. The invariant guarantees that the `parent` field will be `null` for any object references whose type is `FObject` or its subclasses, except `Dir`.

Strategies are illustrated by refactorings applied to the program from Section 2. For pushing down the `parent` field, Law *(move field to superclass)* may be a straightforward option to refactor the program. However, the highlighted statements in Figure 1 clearly prohibit the application of the law, as they make the required provisos invalid. Our aim is to apply other primitive laws that acquire value from this information. The following derivation is related to the method `getParent`, within which `parent` is read. This derivation is illustrative for the example; in fact, this strategy can be generalized for programs in a similar context.

Step 1. Within `FObject`, the value of the `this` identifier obeys the following condition: `this instanceof Dir || !(this instanceof Dir)`. We express this condition by the following `if` statement. Using Law *(assertion condition)*, we can change the body of `getParent` by extracting assertions from each branch.

```
Dir getParent(){
  if (this instanceof Dir){
    assert (this instanceof Dir);
    return this.parent;
  } else{
    assert !(this instanceof Dir);
    return this.parent;
  }
}
```

Step 2. Concerning the first branch, we can introduce a cast to the assignment, using Law *(introduce trivial cast in expressions)*.

```
Dir getParent(){
  if (this instanceof Dir){
    return ((Dir)this).parent;
  } else{
    assert !(this instanceof Dir);
    return this.parent;
  }
}
```

Step 3. In the second branch, we now can use the invariant from `FObject`, defined as `#((FObject-Dir).parent)=0`. It is introduced as an assertion conjoined with the previous one, as follows (translated to Java as `this instanceof Dir || this.parent==null`).

```
Dir getParent(){
```



```

    if (this instanceof Dir){
        return ((Dir)this).parent;
    } else{
        assert (this instanceof Dir || this.parent==null &&
                !(this instanceof Dir));

        return this.parent;
    }
}

```

Step 4. Still in the second branch, we use simple logic rules for simplifying the assertion ($\neg A \ \&\& \ A == \text{false}$).

```

Dir getParent(){
    if (this instanceof Dir){
        return ((Dir)this).parent;
    } else{
        assert (this.parent==null);
        return this.parent;
    }
}

```

Step 5. As stated in the assertion, the program state defines the value read by the assignment (already null before the command), resulting in the following body for *getParent*.

```

Dir getParent(){
    if (this instanceof Dir){
        return ((Dir)this).parent;
    } else{
        return null;
    }
}

```

A similar derivation can be developed in the constructor of *File*, assigning null to parent:

Step 1. Since the command is within *File*, we can introduce the following assertion.

```

File(){
    assert (this instanceof File);
    this.parent=null;
}

```

Step 2. The program invariant $(\text{this instanceof Dir} \ || \ \text{this.parent}==\text{null})$, is then introduced to the assertion.



```
File(){
  assert (this instanceof Dir || this.parent==null &&
          this instanceof File);
  this.parent=null;
}
```

Step 3. The assertion can now be simplified accordingly.

```
File(){
  assert (this.parent==null && this instanceof File);
  this.parent=null;
}
```

Step 4. The command has no effect over the state (`this.parent` possesses a constant value before and after) and no other variable is changed. Hence, the assignment (along with the assertion) can be removed with no impact on the program's behavior.

After the application of this strategy, the program now can be subject to Law *(move field to superclass)*, from right to left. The same could have been done to other similar occurrences of `parent` in the program. Consequently, these can be generalized for any other program presenting this type of invariant. The general sequence of transformations before the actual move operation constitutes a strategy with the aid of the program invariant, to be applied in conjunction with the refactoring tool.

4.3 Other invariants

Similar strategies can be defined for several types of invariants. Some of the invariants we investigated are summarized next:

- **Remove field.** In general automated refactorings only remove fields when they are not used anywhere in the program. A strategy can prepare programs that do not present this property – although removal of the given field is desirable – by replacing all reads from the field to be removed by the correspondent value given by an invariant. For instance, if the invariant `this.newField=this.oldField` is provided for the class declaring the field, and `oldField` is to be removed, we can use the invariant to replace reads from `oldField` by the corresponding expression (`newField`), eliminating writings. This is possible since no other variable depends on this field (all reads have been removed).
- **Replace array field by single variable field.** A field can be declared as an array even though a design assumption defines the field as empty or holding only one element. This is due to planned additions that did not come about, as for example accounts in a bank that were defined with the policy of holding at most one credit card, and this assumption did not change in the future. In this case, we can change its declaration and statements to a single variable, given the invariant on the multiplicity of the field. For instance, an array field `var`, in the presence of an invariant `#this.var=1` on its cardinality, can store this single value on a variable; laws can be applied to change the statement to use the variable (for instance, `this.var[0] = a` becomes `this.var = a`). The reverse transformation (variable to array) can be applied as well.

5 Conclusions and Future Work

In this paper, we describe an approach for automatic refactorings that assumes program invariants for offering more applicability to refactoring tools, avoiding some manual adjustments. Invariants – declared as code annotations – provide semantic information about classes and their fields, which is used to refactor the program in primitive steps – laws of programming – offering a greater degree of applicability.

There are several other open questions. For instance, it is not clear how invariants will be automatically identified by a refactoring tool for the application of specific refactorings. Our intuition is that catalogs of program refactorings could be extended with improvements based on invariants, conditionally applied based on a set of found invariants. Also, other types of invariants must be investigated, in order to establish a more general notion of invariants that can aid automated refactorings. These accomplishments are critical to incorporating invariant-based refactorings in tool support.

We plan also to extend this approach to consider other types of annotations, such as pre- or post-conditions of methods. We believe that these invariants may help more powerful refactorings involving methods, such as the Extract Method refactoring [Fow99]. Other promising research topic is exploring invariants not only for refactorings, but also general evolution transformations (adding a new feature to the program, for example).

In a previous work [MGB05], we propose an approach for refactoring object models (such as UML class diagrams with OCL invariants [B⁺99, W⁺03]) and programs, as show in Figure 3, which is an application of invariant-based refactoring. A refactoring is applied to the object model, and a program in conformance with the model is automatically refactored accordingly. The program refactorings are automatically from the semantic information given by models.

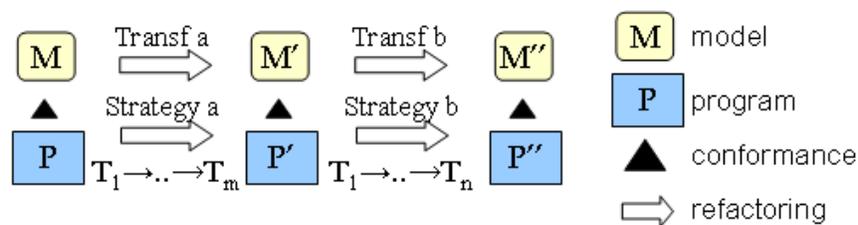


Figure 3: Model-driven Refactoring.

We consider object model refactoring as a composition of primitive semantics-preserving transformations. Each model transformation applied to the model triggers the application of a strategy to the source code. The main idea behind strategies is the assumption that the original program is in conformance with the model, implying that all model invariants are guaranteed to be true in the program. Therefore, the same principle of invariant-based refactoring is applied, in which predefined model transformations provide the original model to which the transformation is applied, so the invariants that are considered true in the program are known in advance. The program transformation is applied independently, although based on the model transformation; this scenario avoids the problems related to round-trip engineering tools, in which programs are generated from models, and vice-versa. This is certainly a useful formal



investigation for modern development methodologies, such as Model-driven Architecture [K⁺03].

References

- [B⁺99] Grady Booch et al. *The Unified Modeling Language User Guide*. Object Technology. Addison Wesley, 1999.
- [B⁺04] Paulo Borba et al. Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, 52:53–100, October 2004.
- [BCC⁺05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. *Software Tools for Technology Transfer*, 2005.
- [ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.
- [FLL⁺02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. *ACM SIGPLAN Notices*, 37(5):234–245, 2002.
- [Fow99] Martin Fowler. *Refactoring—Improving the Design of Existing Code*. Addison Wesley, 1999.
- [HHJ⁺87] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of Programming. *Communications of the ACM*, 30(8):672–686, 1987.
- [K⁺03] Anneke Kleppe et al. *MDA Explained: the Practice and Promise of The Model Driven Architecture*. Addison Wesley, 2003.
- [KMJ02] Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. An Analyzable Annotation Language. In *Proceedings of the 17th OOPSLA*, pages 231–245. ACM Press, 2002.
- [LG01] B. Liskov and J. Guttag. *Program Development in Java*. Addison Wesley, 2001.
- [LM04] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *ECOOP*, pages 491–516, 2004.
- [MGB05] Tiago Massoni, Rohit Gheyi, and Paulo Borba. A model-driven approach to formal refactoring. In *Companion to the OOPSLA 2005*, pages 124–125, USA, October 2005.
- [MT04] Tom Mens and Tom Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.



- [Opd92] William Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
- [W⁺03] Jos Warmer et al. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison Wesley, second edition, 2003.



Generating Requirements Views: A Transformation-Driven Approach

Lyrene Fernandes da Silva* and Julio Cesar Sampaio do Prado Leite**

*Federal University of Rio Grande do Norte – Brazil

**Pontificia Universidade Católica do Rio de Janeiro (PUC-Rio) - Brazil

Abstract. This paper reports the use of transformations based on XML to generate requirements views. A strategy to generate views is defined and scenarios and class diagrams are automatically created from a goal oriented model; the V-graph.

Keywords: Requirements models, views, traceability, transformations, XML.

1. Introduction

During software construction, we may use different types of models and languages such as: scenarios, requirements sentences, lexicons, component models, class diagrams, entity-relationship models, and activity diagrams. These different software representations are necessary because each one of them portrays a different and limited set of characteristics. This is done in order to decrease software complexity and help the engineer to focus on scoped problems.

However, because of the volatility of the requirements, it is necessary to be ready for changes, or evolution. Requirements evolution happens in two ways: during software development, changing from the high abstraction level to the implementation level, i.e., from the requirements to the code; and in order to make the model ready to attend new requirements or fix errors and omissions [19][23]. In both cases, knowing and managing the interactions between requirements (the traceability) is of fundamental importance. As a consequence it is important to decompose and modularize the concerns of the system for two main reasons: (a) the concerns indicate the coupling and cohesion among their components, and (b) it is important in order to analyze the impact of changes between requirements at the same abstraction level and requirements and software artifacts on different abstraction levels [28].

As defined in [33], traceability means the ability to find related requirements in a requirements specification, discovering: the source of the requirements (pre-traceability); the components that implement them (post-traceability); or requirements that affect each other [28]. Traceability is important to manage and to propagate changes in requirements, thus supporting software construction [15]. However, if many different models are used during the construction process, it is necessary to map the information from one kind of model to others, propagating changes, verifying correctness and conflicts among them.



In order to address this problem, in this paper, we present an approach based on transformations to create different views of requirements models. The transformations are defined by using rules and have been implemented in the context of XML (eXtensible Markup Language) technology, by means of XSLT transformations. We exemplify this approach by the creation of rules to transform a goal model, called V-graph, into models: scenarios [23] and class diagrams [4].

This approach was defined in the context of an aspect-oriented requirement modeling strategy [31][32]. In this context, we have considered that using views is extremely important because they can be an alternative way to separate concerns and decrease tangling and scattering problems that occur due to the tyranny of the dominant decomposition [34]. The V-graph model was used because it can represent both non-functional and functional requirements. Furthermore, it explicitly represents the positive and negative interactions between requirements in opposition to use cases, scenarios and requirements sentences which do not tackle the issue of requirements interference. Therefore, we can identify crosscutting concerns by analyzing the interactions among them. In this paper, we present our approach to generate views from V-graph, but we omit the details about “aspects” defined in [31][32] because of the limited space available.

The remainder of this paper is organized as follows. In Section 2, we present the context of this work, the concept of views and the V-graph. In Section 3, we present how we can generate different views by using transformations, the benefits of this strategy and the defined transformation rules to generate scenarios and class diagrams from V-graph models. In Section 4, we present a case study to illustrate the visualization mechanism. In Section 5, we cite some related work. Finally, in Section 6, we present the concluding remarks and guidelines for future works.

2. State of the Art

Considering software development as an evolutionary process, in which the design and the programming are based on the requirements definition and the requirements are continually changing, software evolution happens: during the development process by refining the models created during the requirements definition process into architecture models and code; and during each activity, by making a set of models better, generating different versions of the same model.

In both cases, it is necessary to be able to identify where changes impact, modify and propagate these changes for all the used models. The traceability (or mapping) between models can be supported by a transformation-driven approach [3]. Transformations are interesting in this case because they can be used to:

- Generate views – different total or partial models can be generated from a base model. Partial models can represent the system focusing on different concerns or different viewpoints. Total models can represent the information of a base model using another notation. In both cases, generating views helps software evolution.
- Facilitate analysis – by using different views, the engineer can focus on scoped problems, analyze and modify the modeling in a more effective way. Consistency and completeness checking are facilitated because transformations can generate models with the same information but changing the perspective to analyze it. A proper inspection mechanism, when in place, can point out errors and omissions between views.



- Propagate changes – changes made in a model can be automatically propagated to other models. This decreases the rework, increases the engineer's productivity and guarantees that all models are updated;

On the other hand, it is difficult to use a transformation-driven approach, because it is hard to map one model to others. This is not always possible, usually the concerns represented in a model are not represented in another model, some information can be omitted, and thus it is difficult to guarantee the consistency and completeness of these models.

In the requirements definition process, non-completeness and inconsistencies are more tolerable than in other activities because models created in this stage will be refined by future feedback. These models cannot have all of the information about the solution to the problem and they have to accommodate some conflicts and ambiguities from the domain because these conflicts have to be analyzed and resolved.

For us, the creation of views provides different perspectives of the same model, separating the crosscutting concerns in different ways and offering the requirements engineer different ways to analyze these concerns. In Section 2.1 we present the concept of views used in this work and in Section 2.2 we present the syntax and semantic of the V-graph.

2.1 Views

Views are representations of the overall architecture that are meaningful to one or more stakeholders in the system (IEEE St.1471). Using views is a way of separating different concerns in order to focus on one at a time. Views help understanding and elaborating solutions [33], therefore, they are necessary during all the development process.

In the requirements engineering area, the words *viewpoints*, *views* and *point-of-view* are sometimes used with similar or different meanings [22]. In order to make the view concept clear, in [16], three categories of views are presented:

- Views as opinions (viewpoints) – in the social context, each stakeholder has his premises, priorities and experiences, and they use different ways to deal with the problems. Therefore, it is necessary to know how to compare and negotiate the different opinions or different ways of how to look at the things, for example, what is the opinion of the manager, of the seller and of the buyer about an e-commerce site?
- Views as services (concerns) – the idea of partitioning the system into a set of services that can be connected in different ways provides component-based development. For example, a component for payment of bills, another for security, among others;
- Views as models (perspectives) – in the context of software engineering many techniques based on languages have been proposed in order to partially portray a system, such as entity-relationship models, use cases diagrams and sequence diagrams. Therefore, it is important to detect consistency and completeness problems among these models.

These categories of views are not disjointed categories. Usually, we use **models** (perspectives) to represent **services** (concerns) from the point of view of one or more stakeholders (viewpoint). Furthermore, models, by definition, make some types of information explicit and hide others, so we can have models focusing on functions, data, sequence of activities and so on.

2.2 V-Graph

V-graph is a type of goal model [36]. Goal models represent the functional and non-functional requirements through decomposition trees [25]. V-graph, see Figure 1(a), is defined by **goals**,

softgoals, **tasks** and the following decomposition relationships – **contribution** links (and, or, make, help, unknown, hurt, break) and **correlation** links (make, help, unknown, hurt, break). Each element has a **type** and **topics**. The **type** defines a generic functional or non-functional requirement, for example, Security and Management. The **topic** defines the context of that element, for example, data and communication.

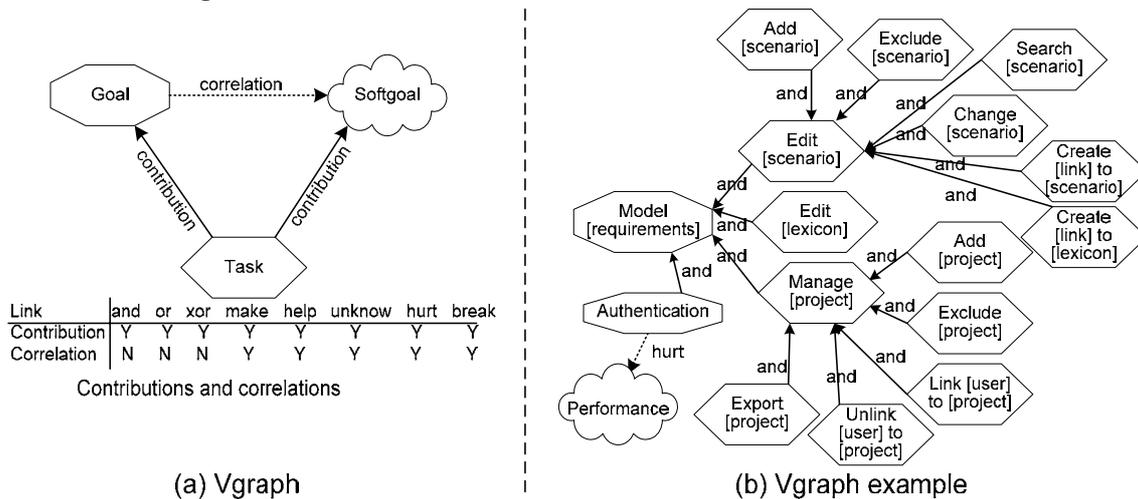


Figure 1. (a) V-graph and (b) a V-graph example

Figure 1(b) portrays a V-graph example. In such figure, we can observe that in order to achieve the goal “Model [requirements]” (“Model” is the type and “requirements” is the topic) the goals “Edit [scenarios]”, “Edit [lexicon]” and “Manage [project]” have to be achieved. The relationships among these four goals are contribution links. The contribution links represent a hierarchy between root (a more abstract element that is father of subelements) and children (an operationalization that can be a leaf or another root of the tree).

V-graph is an interesting model to represent requirements because with it we can consider requirements at three abstraction levels (softgoals, goals and tasks). This is important because in the same model we can represent reasons and operations, the context and how each element contributes to achieving the system goals. Furthermore, there are important results in goal modeling, concerning: how to analyze obstacles to the satisfaction of a goal [18]; how to qualitatively analyze the relationships in goal models; how to analyze variability [14]; how to analyze conflicts among goals through a propagation mechanism of labels [13]; how to identify aspects in goal models [36]; how to derive a feature, state and component model from goal models [30]; and how to provide goal reuse [24] – this last work mentions a composition mechanism used to integrate a goal model and a reusable goal model from a library.

3. Using Transformations to Generate Requirements Views

Views have been used in different activities of software construction because they help the developer to delimit the scope of a problem and thus, its complexity. Therefore, the developer can analyze the correctness and completeness of one concern or a set of concerns at a time. During the requirements definition process, as well as during the design process, i.e., during the elaboration of solutions, it is important that developers be able to obtain different views from a base model in order to facilitate the analyses of the solutions created from different viewpoints and perspectives.

As we defined in Section 2, a view is a representation of the software architecture or of one part of the system's architecture, focusing on one or more concerns, by one or more stakeholders. Figure 2(a) presents, through a features model [8], the variability, considering views as models and as services. This feature model shows that a view represents one or more services using one or more types of models (notation). Therefore, we can create views to the requirements focusing each concern separately (partial views) or in conjunction (total views), using different types of models.

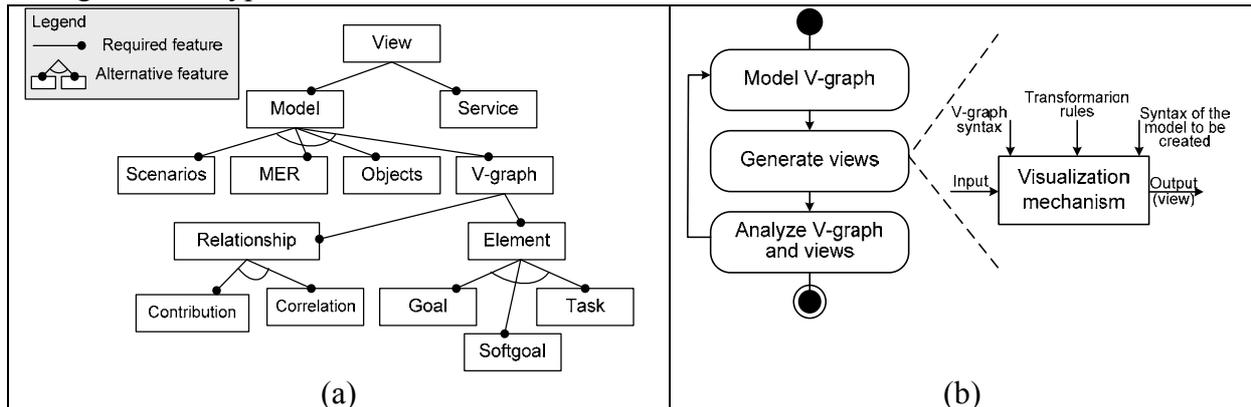


Figure 2. (a) Feature model representing the V-graph views and (b) Visualization mechanism

Therefore, we consider fundamental to have a visualization mechanism in order to facilitate the requirements modeling. An automatic mechanism to generate views can accelerate the modeling process because it decreases reworking and inconsistencies among different requirements models. Figure 2(b) summarizes a modeling process using a visualization mechanism to automatically generate views from V-graph. Such visualization mechanism consists on a transformation component that needs the following sets of information: the syntax and semantic of the source language (in our case, V-graph) and the target language (representation to be generated), and transformation rules. Next subsection deals with how to transform V-graphs into scenarios and class diagrams.

3.1 Transformation Rules

The V-graph is a representation where we can explicitly model functional and non-functional requirements using softgoals, goals and tasks. This is its dominant decomposition manner, an intentional-oriented decomposition. However, the hierarchy and the topics of the V-graph provide new perspectives of the system, based on, for example, situations and data. Furthermore, the relationships between goals, softgoals and tasks provide a perspective of interaction, or traceability. Using this knowledge, we define rules to transform the information from a V-graph into two different models: scenarios [23] and class diagram [4]. Therefore, we provide requirements views that help “combating” V-graph’s dominant decomposition.

Transforming V-graph into Scenarios

Scenarios are an interesting topic to the software engineering community [35][29]. The scenario-driven software development is based on the concept that using the problem's language (user’s domain) is really beneficial for the interaction and communication between users and developers. Scenarios are common situations to the users [31]. They have to take

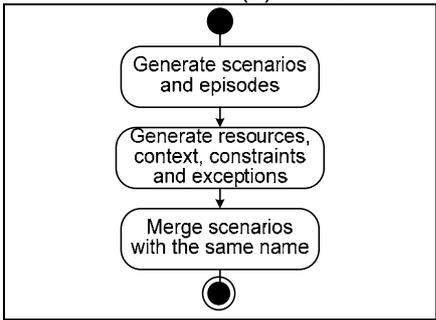
into account the usability, enable the comprehension of the domain and problem and help unifying criteria, the sorting of details and user training [7].

Each scenario describes, through semi-structured natural language, a specific situation of the application or of the domain, focusing on behavior. Scenarios can be detailed and used as design, in order to help programming. There are many representation proposals for scenarios, informal representations in free text [7] or formal representations [17]. We opted for an intermediate representation: it facilitates the comprehension using natural language and forces sorting of the information by using a well-defined structure, proposed in [23]. In order to transform the information in V-graph into a scenarios model some steps are followed:

1. First, we generate scenarios and episodes by using the goal tree hierarchy: each node (goals and tasks) of the tree that is not a leaf generates a scenario; each subgoal and subtask that is a leaf generates an episode of that scenario; and each subgoal and subtask that is not a leaf generates an episode with a reference (link) to a scenario; contributions “or” generate optional episodes.
2. Second, we generate resources, constraints and context. Each topic of goal and task that generated episodes without references to any scenario is a resource of that scenario; each softgoal negatively related (correlation or contribution) to goal/task that generated a scenario generates an exception into that scenario; each softgoal negatively related (correlation or contribution) to goal/task that generated an episode (without reference to any scenario) generates a constraint into that episode; each goal related (contribution) to task that generated a scenario generates the context into that scenario.
3. Next, we merge scenarios with the same title, because it is possible there be more than one goal or task with the same name in V-graph in order to facilitate the visualization of the tree of goals.

Table 1 summarizes the transformation process from V-graph to Scenarios and Section 4 presents an example of scenarios model generated by using this process.

Table 1. (a) Transformation process and (b) transformations: V-graph \rightarrow Scenarios

 <p>(a)</p>	<table border="1"> <thead> <tr> <th>V-graph</th> <th>Scenarios</th> </tr> </thead> <tbody> <tr> <td>Goals, tasks</td> <td>Scenarios and episodes in accordance of the hierarchy of goals and tasks</td> </tr> <tr> <td>Softgoals negatively correlated</td> <td>Exceptions or constraints</td> </tr> <tr> <td>Goals of tasks that generated scenarios</td> <td>Context</td> </tr> <tr> <td>Topics of goals and tasks that generated episodes without references to scenarios</td> <td>Resources</td> </tr> </tbody> </table> <p>(b)</p>	V-graph	Scenarios	Goals, tasks	Scenarios and episodes in accordance of the hierarchy of goals and tasks	Softgoals negatively correlated	Exceptions or constraints	Goals of tasks that generated scenarios	Context	Topics of goals and tasks that generated episodes without references to scenarios	Resources
V-graph	Scenarios										
Goals, tasks	Scenarios and episodes in accordance of the hierarchy of goals and tasks										
Softgoals negatively correlated	Exceptions or constraints										
Goals of tasks that generated scenarios	Context										
Topics of goals and tasks that generated episodes without references to scenarios	Resources										

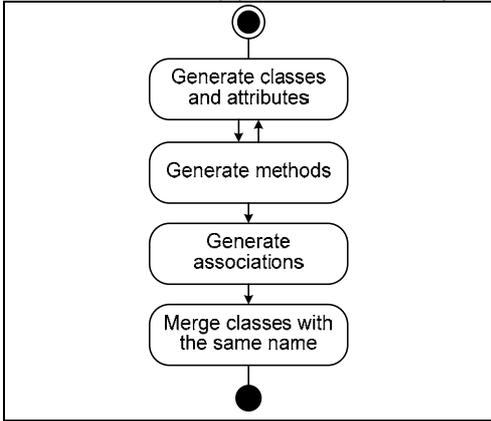
Transforming a V-graph into a Class Diagram

Objects are executable entities, instances of a class that defines its attributes and services. Object or class models, usually, are used when we want to adopt the object-oriented paradigm in the design and programming activities. However, these models can be used during the requirements definition process to represent data and functionalities [33]. In this sense, we derived class diagrams from V-graph models. In order to transform the information in V-graph into a class diagram, we followed the general pattern described below.

1. First, we use the hierarchy of goals and tasks to define classes and attributes: each topic of goal/task that is not a leaf generates a class whose name is the topic name; and each topic of goal/task that is a leaf generates an attribute in class generated by its parent node.
2. Second, we define the methods of the classes: every goal/task is method of classes whose name is its topic; each goal/task that is a leaf generates a method into class generated by its parent node; if one of the methods refers any topic different of class name then this topic is an attribute of that class.
3. Next, we generate the relationships between classes: classes refer at least one similar method are associated; each correlation or contribution between elements that generate classes generates an association into the class diagram;
4. Finally, classes with same name are merged.

Table 2 summarizes the process to transform V-graph into Class Diagram and Section 4 presents an example of class diagram generated by using this process.

Table 2. (a) Transformation process and (b) transformations: V-graph \rightarrow Class diagram

 <pre> graph TD Start(()) --> A[Generate classes and attributes] A <--> B[Generate methods] B --> C[Generate associations] C --> D[Merge classes with the same name] D --> End(()) </pre> <p style="text-align: center;">(a)</p>	<table border="1"> <thead> <tr> <th data-bbox="719 878 1062 909">V-graph</th> <th data-bbox="1062 878 1417 909">Class diagram</th> </tr> </thead> <tbody> <tr> <td data-bbox="719 909 1062 1003">Topic</td> <td data-bbox="1062 909 1417 1003">Classes and attributes in accordance of the hierarchy of goals and tasks</td> </tr> <tr> <td data-bbox="719 1003 1062 1039">Goals, tasks</td> <td data-bbox="1062 1003 1417 1039">Methods</td> </tr> <tr> <td data-bbox="719 1039 1062 1299">Contributions, correlations, and goals/tasks that refer similar topics</td> <td data-bbox="1062 1039 1417 1299">Associations</td> </tr> </tbody> </table> <p style="text-align: center;">(b)</p>	V-graph	Class diagram	Topic	Classes and attributes in accordance of the hierarchy of goals and tasks	Goals, tasks	Methods	Contributions, correlations, and goals/tasks that refer similar topics	Associations
V-graph	Class diagram								
Topic	Classes and attributes in accordance of the hierarchy of goals and tasks								
Goals, tasks	Methods								
Contributions, correlations, and goals/tasks that refer similar topics	Associations								

3.2 Implementation

We implemented this strategy using XML (*eXtensible Markup Language*) and XSLT (*eXtensible Stylesheet Language Transformation*). V-graph syntax was defined using a DTD (*Document Type Definition*). The visualization mechanism, i.e., the transformations, was programmed in XSLT. Therefore, a V-graph model (in XML) is the input to the visualization component and the outputs are scenarios (in HTML) and class diagrams (in Dot). Figure 3 shows how we used XML and XSLT to implement our strategy. The Dot format and the GraphViz application [16] are used to create graphic representations to V-graph and class diagrams.

The choice for XML based transformations was due to the characteristics of our proposal, models written in XML, and due to the characteristic of our transformation rules. In our case, the transformations were localized and direct, and as such the XSLT mechanism was sufficient. Of course if more complex rules were necessary, we would have to use a more robust transformation platform. However it is important to stress that despite its simplicity, we can easily implement different representations and layouts for the information described in XML with its structure defined in a DTD. Furthermore, XML can easily be read and changed for different applications.

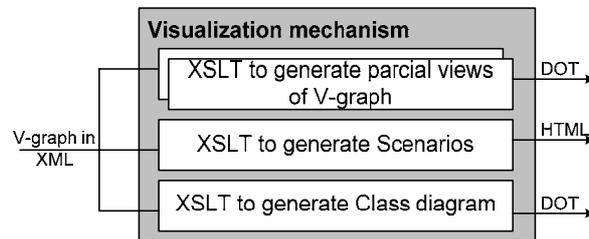


Figure 3. XML and XSLT used to transform requirements views

4. Case Study

This section presents part of our case study. The complete case study has four goal models: a goal model for an information system that helps writing scenarios and lexicon [32]; a goal model for Security; a goal model for Reliability; and a goal model for Persistence. The V-graph illustrated in Figure 1(b) is the part of this case study we have used in this paper to demonstrate our approach. On the right side of Figures 4 e 5 we show this same V-graph. The octagons are goals and the hexagons are tasks. Pointed links are correlations and the other links are contributions. Each goal and task has at least one Type and zero or more Topics (bracketed text). On the left side of Figures 4 e 5, we portray the created views: scenarios and class diagram.

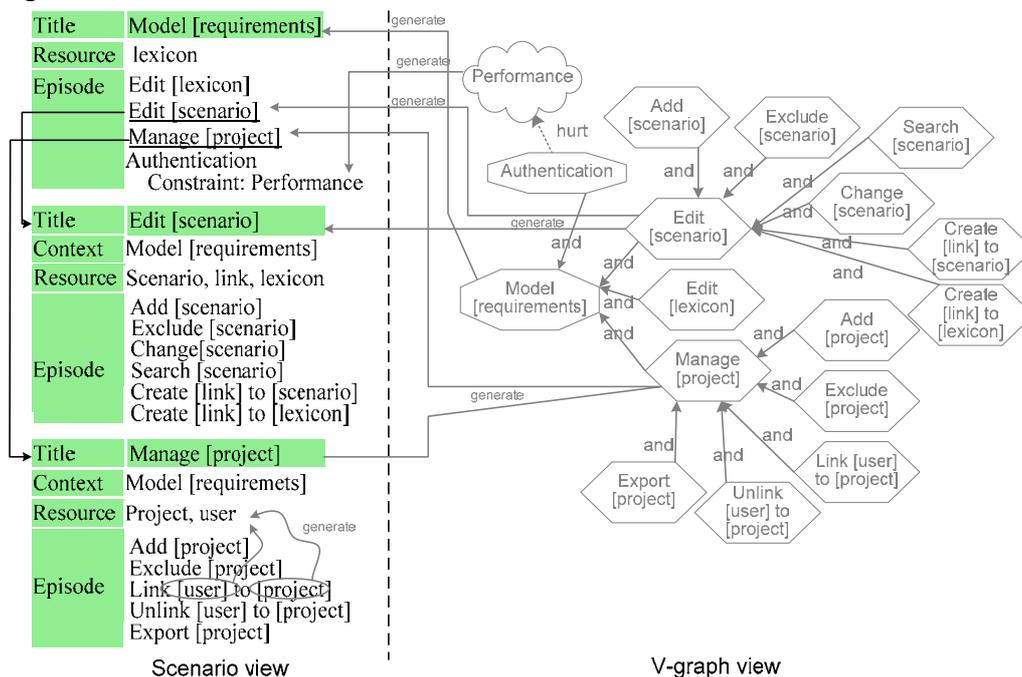


Figure 4. Example of the scenario view

Figure 4 portrays an example of scenarios generated from the V-graph. In this example, there are three scenarios with the titles “Model [requirements]”, “Edit [scenario]” and “Manage [project]”; they were derived from goals with similar names. In the scenario “Manage [project]”, there are two resources (project and user) generated from topics of their episodes; the context attribute is generated from the goal “Model [requirements]”; the attribute episode is generated based on the goals/tasks that decompose the goals “Model

[requirements]”, “Edit [scenario]” and “Manage [project]”; and each underlined episode indicates a relationship with another scenario.

Figure 5 portrays an example of class diagram to the V-graph (in Figure 1b). Each goal/task that is not a leaf (and has topics) generates a class into the class diagram. Children of goal/task that do not generate classes (and have topics) generate attributes related to the class generated by their goals/tasks parent. Relationships between classes are generated from the contribution links between goals/tasks that generate those classes. In Figure 5 we can observe the classes “requirement”, “project” and “scenario”, their attributes and operations; the relationships between them are generated based on the relationships between tasks/goals whose topics are source to these classes.

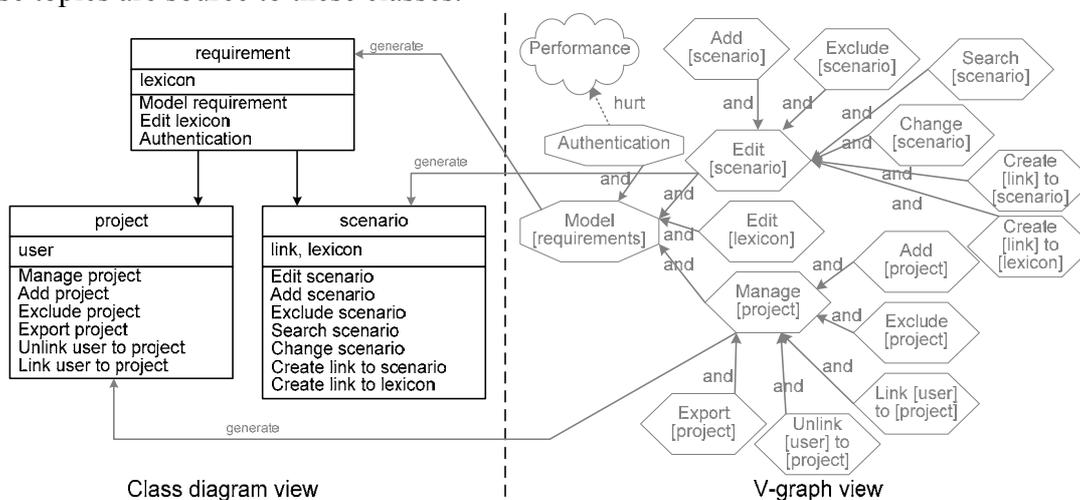


Figure 5. Example of class diagram

This case study is available in [32]. The complete modeling has 9 goals, 105 tasks, 12 softgoals, 7 correlations and 173 contributions. After the transformation process applied to this case study, 40 scenarios and 14 classes were generated into the scenarios view and class diagram view, respectively. Although the class diagram does not have the types of relationships and cardinality, the models created help the engineer to analyze the domain from the data perspective in opposition to the intentional perspective of the V-graph. Scenarios and class diagrams represent two other dominant ways to separate concerns, therefore by using our approach the engineers have these views without having to create them manually. Furthermore, any change made in the V-graph can be automatically propagated to the other models.

5. Related Work

Although using different types of models helps managing the complexity during software modeling, it causes tangling and scattering of concerns, making it difficult to maintain every model consistent and updated. Therefore, integration mechanisms are necessary in order to integrate services and models as well as to integrate opinions (conflicts resolution). This paper focused on the integration of models using a transformation-driven approach.

Software transformation has been a central topic in different software related areas. In the early seventies/eighties several researchers believed it to be central to the idea of automatic programming and several program transformation initiatives were initiated, notably



the Irvine Transformation Catalogue [11]. Also in the area of software reuse, the idea of software transformation was particularly successful, for instance [1] and the approaches on product-line [2] and the Draco approach to software construction [21][26][30]. The use of transformations, in this type of context, requires a more powerful mechanism, since the control structure is not straightforward and a strict discipline to help the validation of the complex rewrite rules is necessary.

There are also many approaches less complex which have been used to transform requirements models into other requirements models or design models. Many of them are based on natural language that process or consider the structure of the source language to identify the constructs of the target language, such as:

In [6], a process to generate ontology from LEL is defined. This approach is based on transformations but it is only semi-automatic. In [5], an approach has been defined to generate activity diagrams from use cases and after that to transform these diagrams into Pres, a formal notation that permits verification. Therefore, this approach enables the enrichment of the use case model and the production of more precise and complete requirements. In [12], an automated approach to transform feature models into the class diagrams is defined.

In [9], a process to integrate RNFs and RFs is defined. This approach uses the constructs of MER, of class diagram and of lexicon extended language (LEL) [20], in order to make this integration. Such integration process is based on mapping the RNFs specified in LEL into the MER and into the class diagram. However, this mapping is not based on transformations, it is based on the analysis of the information in the LEL, MER and in the class diagram.

In [27], an integration framework of models (modeling methods) is defined. This framework determines the specification of: (1) style – defines the notation; (2) work plan – specifies the activities, strategies and processes to define a view; (3) domain – indicates the domain area; (4) specification – the development method; and (5) work report – indicates the state and history of the modeling. The information described in (1) e (2) is abstract information; they can be applied to every instance of the same type of model whereas information in (3), (4) and (5) of this framework is specific to each instance of the model. The main goal of this work is to give support for consistency checking among different models and managing inconsistencies, facilitating the reuse of information on how to map one representation into others. This framework inspired us to define informally the information described in (1) and (2) in order to specify the transformations from V-graph into scenarios and class diagrams, as we have shown in Section 3.2.

6. Final Remarks

In this paper, we present a visualization mechanism used to generate requirements models. This mechanism is transformation-driven. We created some transformation rules in order to automatically generate scenarios and class diagrams from the V-graph model. Using transformations during the requirements definition helps us make the trace among the different models used. It facilitates modeling because consistent models are generated and changes are automatically propagated. Consequently, these transformations help software evolution.

The results that we have had using this approach have been satisfactory because we consider that generated views help the engineer analyze the system. However, such views cannot be considered complete models, but initial models that help the engineers because they



do not have to begin the modeling from scratch. Future work involves: making better transformation rules in order to obtain more detailed models; defining transformation rules to both directions, V-graph \rightarrow (scenarios and class diagrams) and (scenarios and class diagrams) \rightarrow V-graph; creating a verification mechanism to report inconsistencies and omissions into each type of view; and also tools are extremely necessary to support the edition of any of these models. Furthermore, it is necessary to plan experiments in order to validate our approach at the requirements definition stage and to evaluate what is the impact of using it during the entire software development process.

Currently, we are working on the definition of the transformation rules to generate the system architecture from the requirements definition. This work is part of our aspect-oriented approach to model requirements [31][32]. When taking crosscutting concerns into account, the visualization approach presented in this paper is equally important because using views is an alternative way to separate crosscutting concerns, facilitating the tasks of modeling, analysis, traceability and software evolution.

7. References

1. D. Batory, S. Dasari, B. Geraci, V. Singhal, M. Sirkin, J. Thomas. Achieving reuse with software system generators. In: IEEE Software, September-1995, pp. 89-94.
2. D. Batory, R. Lopez-Herrejon and P. Martin. Generating Product-Lines of Product-Families. In: Automated Software Engineering Conference, 2002.
3. I. Baxter. **Transformational Maintenance by Reuse of Design Histories**, Ph.D. Thesis, Information and Computer Science Department, University of California at Irvine, Nov. 1990, TR 90-36.
4. G. Booch, J. Rumbaugh and I. Jacobson. **The Unified Modeling Language User Guide**. Addison-Wesley, 1999.
5. R. Boudour and M. Kimour. Model Transformation for Requirements Verification in Embedded Systems, In: **Asian Journal Informational Technology**, 4 (11): 1012-1019, 2005.
6. K. Breitman¹, J. Leite. Lexicon Based Ontology Construction. In: Lecture Notes in Computer Science 2940- Editors: C. Lucena, A. Garcia, A. Romanovsky, et al., ISBN: 3-540-21182-9, Springer-Verlag Heidelberg, February 2004, pp.19-34.
7. J. Carroll et al. d'etre: capturing design history and rationale in multimedia narratives. In: HUMAN FACTORS IN COMPUTING SYSTEMS (CHI94), Boston-USA, ACM Press, 1994, p. 192-197.
8. K. Czarnecki and U. Eisenecker. **Generative Programming: Methods, Tools, and Applications**, Addison-Wesley, 2000.
9. L. Cysneiros, J. Leite and J. Neto. A Framework for Integrating Non-Functional Requirements into Conceptual Models. **Requirements Engineering Journal**, Vol. 6, No. 2, p. 97-115, 2001, Springer-Verlag London Limited.
10. Draco - Software Reuse, Domain Analysis and Draco Information. Available at: <http://www.bayfronttechnologies.com/102draco.htm>. Accessed on: Mar, 7th, 2006.
11. M. S. Feather. A Survey and Classification of some Program Transformation Approaches and Techniques. In IFIP 87, pages 165-195, 1987.
12. F. García, M. Laguna, Y. González-Carvajal and B. González-Baixauli. Requirements variability support through MDD and graph transformation. Submitted to Elsevier Preprint, 2005.
13. P. Giorgini, J. Mylopoulos, E. Nicchiarelli and R. Sebastián, Reasoning with goal models, Proceedings of the 21st International Conference on Conceptual Modeling, 2002, pp. 167-181.
14. B. Gonzáles, M. Laguna and J. Leite, "Visual variability analysis with goal models", Proceedings of IEEE International Symposium on Requirements Engineering (RE'04), Japan, 2004, pp. 38-47.
15. O. Gotel, and A. Finkelstein. An analysis of the requirements traceability problem. In: PROC. OF THE FIRST INTERNATIONAL CONFERENCE ON REQUIREMENTS ENGINEERING



- (ICRE'94), IEEE Computer Society Press., 1994. p. 94-101.
16. GRAPHVIZ. Available at: <http://www.graphviz.org/>. Accessed on: Mar, 7th, 2006.
 17. P. Hsia et al. Formal Approach to Scenario Analysis. **IEEE Software**, vol. 11, No. 2, 1994. p. 33-41.
 18. A. Lamsweerde and E. Letier, "Handling obstacles in goal-oriented requirements engineering", *IEEE Transaction Software Engineering*, 26(10):978–1005, 2000.
 19. M. Lehman. Laws of software evolution revisited. **Lecture Notes in Computer Science**, Vol. 1149, 1996. p.108-120.
 20. J. Leite and A. Franco. O Uso de Hipertexto na Elicitação de Linguagens da Aplicação. In: ANAIS DE IV SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 1990. p. 134–149.
 21. J. Leite, **M. Sant'Anna, F. Gouveia**. Draco-PUC: A Technology Assembly for Domain-Oriented Software Development, International Conference on Software Reuse 1994.
 22. J. Leite. Viewpoints on Viewpoints. In: ACM Joint Proceedings of the SIGSOFT'96 Workshops, ACM Press, 1996. p. 285-288.
 23. J. Leite et al. Enhancing a requirements baseline with scenarios. In: PROC. OF THE THIRD IEEE INTERNATIONAL SYMPOSIUM ON REQUIREMENTS ENGINEERING (RE'97), IEEE Computer Society Press, 1997. p. 44-53.
 24. J. Leite, Y. Yu, L. Liu, E. Yu and J. Mylopoulos, "Quality-Based Software Reuse", Proceedings of the CAiSE 2005-LNCS 3520, 2005, pp. 535-550.
 25. J. Mylopoulos, L. Chung, and B. Nixon, "Representing and using nonfunctional requirements: A process-oriented approach", *IEEE Transactions on Software Engineering*, 18(6):483–497, June 1992.
 26. J. Neighbors. The Draco Approach to constructing Software from reusable components. In: *IEEE Trans. on Software Engineering*, vol. SE-10, No.5, pp.564-574, September-1984.
 27. B. Nuseibeh. Crosscutting requirements. In: PROC. OF THE 3RD INTERNATIONAL CONF. ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT ([AOSD 2004](#)), Lancaster-UK, 2004. p. 3-4. ISBN:1-58113-842-3.
 28. W. Robinson, S. Pawlowski and V. Volkov. Requirements Interaction Management. **ACM Computing Surveys**, Vol. 35, No. 2, 2003. p. 132-190.
 29. C. Rolland et al. A proposal for a scenario classification framework. **Journal of Requirements Engineering**, Vol. 3, Springer Verlag, 1998. p. 23-47.
 30. M. Sant'anna, J. Leite and A. Prado. A Generative Approach to Componentware. In: Proc. of the Workshop on Component-based Software Engineering, ICSE'20, Kyoto, Japan, April 1998.
 31. L. Silva, J. Leite. An Aspect-Oriented Approach to Model Requirements. In: RE'05 DOCTORAL CONSORTIUM in conjunction on the 13th IEEE International Requirements Engineering Conference, Paris-France, 2005.
 32. L. Silva. **An Aspect-Oriented Strategy to Model Requirements**. Rio de Janeiro, 2006. 220p. PhD Thesis on Software Engineering - PUC-Rio. In Portuguese.
 33. I. Sommerville. **Software Engineering**, Ed. 6th, Addison- Wesley, 2000.
 34. P. Tarr, et al. "N Degrees of Separation: Multi-Dimensional Separation of Concerns". In: PROC. OF THE 21st Int'l Conf. on Software Engineering (ICSE'99), 1999. p. 107-119.
 35. K. Weidenhaupt et al. Scenario Usage in system development: current practice. **IEEE Software**, Vol. 15, No. 2, 1998. p. 34-45.
 36. Y. Yu, J. Leite and J. Mylopoulos, "From goals to aspects: discovering aspects from requirements goal models", Proceedings of IEEE International Symposium on Requirements Engineering (RE'04), Japan, 2004, pp. 38-47.
 37. Y. Yu, J. Mylopoulos, A. Lapouchnian, S. Liaskos and J. Leite, "From stakeholder goals to high-variability software design", Internal report, 2005.
 38. L. Zorman. **Requirements Envisaging through utilizing scenarios – REBUS**. 1995. Ph.D. Dissertation, University of Southern California.



A MDE-Based Approach for Developing Multi-Agent Systems

Viviane Silva*, Beatriz de Maria** and Carlos Lucena**

*Departamento de Sistemas Informáticos y Programación, UCM, Spain

** Departamento de Informática, PUC-Rio, Brazil

***Abstract.** This paper focuses on the development of multi-agent systems based on a model driven engineering approach. Our goal is to cope with the traceability between design and implementation models and with the always changing characteristics of such systems.*

Keywords: model driven architecture, multi-agent systems, transformation, evolution

1 Introduction

The development of multi-agent systems (MAS) has rapidly increased in the last few years. Modeling languages [13][17][22], platforms [16][18], methodologies [2][4][15] and some others MAS modeling and implementing techniques have been proposed with the purpose of helping the developers in building such systems. Although several approaches concern with modeling and implementing MAS, only few accomplish the tracking between design models and implementation code.

While dealing with MAS, the refinement of design models into implementation code becomes especially difficult since it is necessary to deal with different paradigms during the system development. The agent-oriented paradigm is used while modeling the systems but, frequently, those systems are implemented by using the object-oriented paradigm (OO). Since agents and objects have different properties and characteristics (for instance, agents are autonomous and goal-oriented entities that execute plans in order to achieve their goals and, different from objects, do not need external stimulus to execute and can ignore requests), the transformation from agent-oriented design models into OO code is not simple. To try to assist the implementation of MAS, several OO platforms, architectures and frameworks such as [18][16] have been proposed. Although such approaches satisfactory provide support for the implementation of the MAS, they failed in providing the tracing between the design models and the implementations. The numerals MAS modeling language, methodologies and platforms deal with different agent properties and characteristics what directly impact in the traceability.

Another important concern that mostly affects the development of MAS is the always changing characteristic of MAS applications and techniques. Since a number of fundamental questions about the nature and the use of the agent-oriented approach are still being answered, important techniques features and also applications requirements are still evolving.



This paper focuses on helping the MAS developers to cope with (i) the mapping between design models and implementation models and with (ii) the always changing characteristics of the MAS applications and techniques. In order to achieve these two goals, we use a model driven engineering (MDE) approach [11] for developing MAS. Being our proposal a MDE approach we specify (i) the modeling languages being used to describe the source and target models, (ii) both models, (iii) the transformations, and (iv) some guidelines to cope with models evolution [11].

The traceability between MAS design models and OO implementation models will be illustrated by the use of the multi-agent system modeling language called MAS-ML [17][19] (the source modeling language), the agent society framework called ASF [18] and the UML modeling language [14] (the target modeling language). The MAS-ML design models (the source models) will be transformed into UML implementation models (the target models) by instantiating the ASF framework according to the application characteristics.

Our second goal is accomplished by demonstrating that the transformation rules can be adapted to the evolution of both applications and techniques begin used in the transformation. Such adaptation is facilitated due to the low coupling between design and implementation models. Design models are independent of the platform / framework being used to implement the system, i.e., design models does not concern with any characteristic of the implementation technique. In addition, the framework is also defined completely independent of the modeling language being used in the design models.

The paper is organized as follows. In Section 2 we present the MAS techniques being used in the paper: MAS-ML and ASF. Section 3 introduces the transformation process and Section 4 provides some guidelines to embrace the evolution of applications and techniques. Section 5 describes some related work and Section 6 draws the conclusions and discusses future work.

2 Multi-Agent System Techniques

The development process of complex and large-scale systems, such as MAS, involves the construction of different models based on a variety of requirements. The transformation of a system specification into models and of these models into code is usually accomplished in a non-organized way that is not easily adaptable to technology changes. In fact, it is possible to find different modeling languages, methodologies and platforms for modeling and implementing MAS but it is hard to find in the literature approaches that trace the design models into code. Therefore, we propose in this paper a top-down MDE approach that traces MAS-ML design models into ASF object-oriented code.

2.1. The MAS-ML Modeling Language

MAS-ML is a platform independent modeling language that extends UML incorporating agent-oriented abstractions (such as agents, organizations, environments and roles), their properties (such as goals, plans, actions, beliefs and protocols) and relationships (such as play, inhabit and ownership). Although MAS-ML uses agent and object-oriented abstractions, MAS-ML does not restrict the implementation of its models to a specific implementation platform.

Figure 1 illustrates an important part of the MAS-ML metamodel that presents the agent and object-oriented abstractions defined in the metamodel and the possible relationships among them. By using MAS-ML it is possible to model, for instance, the roles that agents can

play (by using a static diagram defined in MAS-ML called organization diagram) and agents achieving their goals while executing their plans (by using the extended UML sequence diagram defined in MAS-ML).

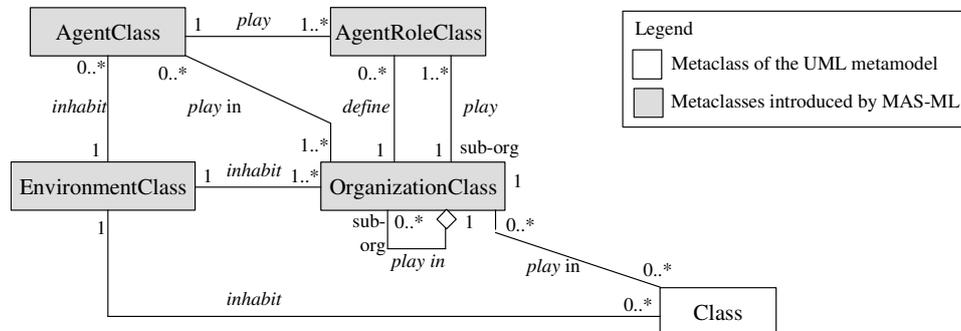


Figure 1. Part of the MAS-ML metamodel

2.2. The ASF Framework

The goal of ASF is to help designers implementing MAS by using the object-oriented paradigm. The framework defines a set of OO models where each model represents a MAS entity type. The set of OO classes and relationships defined in each module makes possible the implementation of the structural aspects of MAS entities (its properties and relationships with other entities) and also the dynamic ones (their behavior).

Figure 2 shows a UML class diagram modeling all ASF classes grouped by modules. The modules that are used to instantiate agents (delimited by a continuous rectangle), organizations (defined by the dotted enlase) and agent roles (marked by the hatched rectangle) are complex modules since they group several classes to represent all the properties of these entity types. The module that corresponds to environments (hatched circle) is simple represented by one class because the properties of this entity can be directly represented as attributes and methods.

In order to use ASF to implement a MAS application, it is necessary to instantiate the framework by extending the defined modules. The extensions should be made according to the entities characteristics defined in the application being implemented. For instance, to implement application agents by using ASF it is necessary (i) to create an OO class extending the *Agent* abstract class defined in the ASF agent module to be used to instantiate the agents, (ii) to create OO classes by extending the *Plan* and *Action* classes to implement the plans and actions of the agents, (iii) to implementing the constructor method of the new agent class to create the (instances of the) beliefs, goals and plans and also (iv) to relate the agent instances to the roles that they will play, the organizations where they will play such roles and also to the environments that they will inhabit. Similar steps could be followed in order to implement the organizations, roles and environments defined in the MAS application.

3 The Transformation Process

In this section we describe the transformation process used to refine MAS-ML design models into UML implementation models that instantiate ASF. The transformations were defined by using the Atlas Transformation Language (ATL) [9]. An ATL transformation program is composed of rules (described in ATL) that define how source model elements are matched and navigated to create and initialize the elements of the target models. Besides the rules, an ATL program receives as input (i) the metamodel of the source model (the MAS-ML

metamodel), (ii) the source model itself (a MAS-ML model) and (iii) the metamodel of the target model (the UML metamodel). The program checks the source model according to its metamodel and, by using the transformation rules (MAS-ML2ASF rules), transforms the source model into the output target model (UML model) that is compliant with the target metamodel. In MDE [11], transformation processes based on the translations between metamodels are called *language translations*. Figure 3 illustrates the inputs and the output of the ATL program that transforms MAS-ML models into UML models by using the MAS-ML2ASF rules.

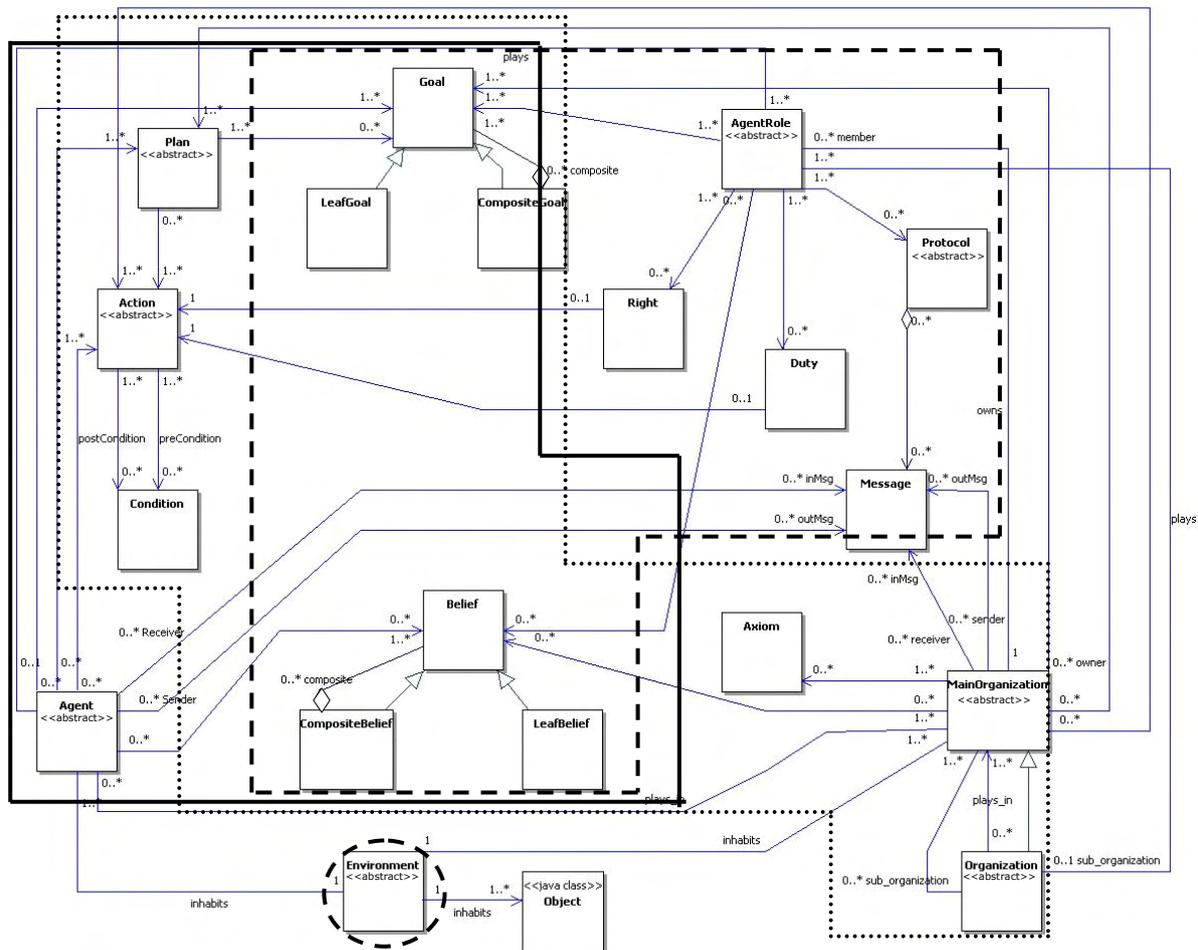


Figure 2 The ASF classes and relationships

The set of rules used by the ATL program to transform a MAS-ML model into a UML model instantiates the ASF framework according to the application features modeled in MAS-ML. The set of MAS-ML2ASF rules is composed of (i) one general rule that generates all the classes defined in ASF (and illustrated in Figure 2); (ii) five rules to transform the five entity types found in MAS-ML models (and defined in the MAS-ML metamodel depicted in Figure 1); and (iii) three other rules to generate the concrete plans, actions and protocols that extends the abstract classes *Plan*, *Action* and *Protocol* defined in ASF (also illustrated in Figure 2).

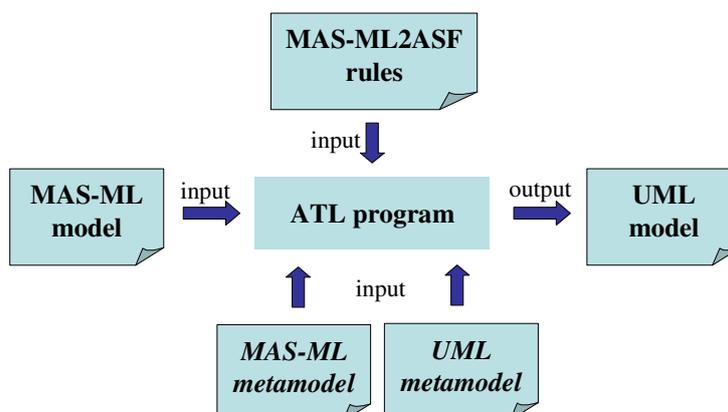


Figure 3 The ATL transformation process

3.1. The Transformation Rules

Due to page limitation, we concentrate on demonstrating the transformation of agents modeled using MAS-ML into OO classes by instantiating ASF. The following sections describe the MASML2ASF rules used by the ATL program that contemplates such transformation.

3.2.1 The creation of the concrete agent class

For each distinguish agent class modeled in a MAS-ML structural diagram, one object-oriented agent class that extends the abstract ASF class called *Agent* must be created. Therefore, we have implemented a rule that is executed for each agent defined in MAS-ML. Such rule, illustrated in the box below, is responsible for creating the agent class and for implementing the constructor method (detailed in Section 3.2.2).

The new classes are created with the same name of the agent classes modeled in MAS-ML models. Associated with each agent class, we define a generalization relationship in order to model the new agent class extending the abstract *Agent* class defined in ASF. Due to such generalization, the new agent class inherits all the attributes (and relationships) defined in the abstract class.

The set of features of the new class is composed of (i) the constructor method, (ii) methods for instantiating goals, beliefs, plans and actions, and (iii) methods for creating the associations between agent instances and the role instances to be played, and the associations between the agent instances and the environment instances.

```
rule Agent {
  from
    c : masml!AgentClass
  to
    -----
    -- Creating the agent class
    outAgent : uml!Class (
      name <- c.name, --the name of the new class is equal to the name of the agent
      isAbstract <- false, --the new class is not an abstract one
      generalization <- genAbstractAgent, --creating the generalization relationship
      feature <- Set{constMethod, createBeliefs, createGoals, createPlans, createActions,
        createInhab, createPlays}), --creating the set of structural and behavioral features
    -----
    -- Creating the generalization relationship between Agent and UserAgent
    genAbstractAgent : uml!Generalization (
      parent <- c.superClass,
      child <- outAgent
    ),...}
}
```



3.2.2 The implementation of the constructor method

The constructor method of agent classes invokes other methods that are used to instantiate the agent properties (goals, beliefs, plans and actions) and to set the relationships between the agents instances, their roles, and environments. Therefore, the body of this method is very simple.

```
-----
-- Constructor Method
constMethod : uml!Method (
  specification <- operationAgent,
  body <- 'createBeliefs(); createGoals(); createPlans(); createActions(); createPlays();
        createEnv(); createPlays();'
),
operationAgent : uml!Operation (
  isAbstract <- false,
  specification <- 'UserAgent()'
),
),
```

The instantiation of the agent properties are exemplified by describing the instantiation of beliefs and plans. The beliefs and goals of an agent are created by instantiating the ASF classes *Belief* and *Goal*, respectively. Therefore, the body of the method responsible for generating the beliefs creates one new belief instance from the *Belief* class for each belief defined by the agent classes of the MAS-ML model being transformed.

The plans and actions of an agent are created by instantiating the classes that correspond to the plans and actions themselves. Those classes are extensions of the *Plan* and *Action* classes as demonstrated in Section 3.2.3.

```
-----
-- Instantiating the agent beliefs
createBeliefs : uml!Method (
  specification <- operationCreateBeliefs,
  body <- c.beliefs -> iterate (e; body : String = '' | body+
    Belief newBelief = null;
    beliefs = new Vector();
    newBelief = new Belief('+e.type+', '+e.value+'); -- instantiating a belief
    beliefs.add(newBelief);')
),
operationCreateBeliefs : uml!Operation (
  isAbstract <- false,
  specification <- 'void createBeliefs()'
),
),
-----
-- Instantiating the agent plans
createPlans : uml!Method (
  specification <- operationCreatePlans,
  body <- c.plans -> iterate (e; body : String = '' | body+
    Plan newPlan = null;
    plans = new Vector();
    newPlan = new '+e.name+'(); -- instantiating a plan
    plans.add(newPlan);')
),
operationCreatePlans : uml!Operation (
  isAbstract <- false,
  specification <- 'void createPlans()'
),
),
```

After instantiating the agent properties, it is necessary to relate the agent instances with the roles to be played, the organization where the roles will be played and the environments that they will inhabit. In order to do so, two methods were created. Both methods are called by the agent constructor method every time an agent instance is created.

```
-----
-- Environment
createInhab : uml!Method (
  specification <- operationInhab,
  body <- c.inhRel -> iterate (e; body : String = '' | body+
    this.environment = new '+e.env.name+'();) --environment
),
operationInhab : uml!Operation (
  isAbstract <- false,
  specification <- 'void createEnv()'
),
```



```

),
-----
-- Roles being played
createPlays : uml!Method (
  specification <- operationPlays,
  body <- c.playRelAg -> iterate (e; body : String = '' | body+
    //roles being played
    AgentRole newRole = null;
    rolesBeingPlayed = new Vector();
    newRole = new '+e.role.name+'();-- instantiating the role
    newRole.setAgent(this);
    newRole.setOrganization('\'+e.org.name+'\');-- associating with the organization
    rolesBeingPlayed.add(newRole);
    // organizations where is playng roles
    MainOrganization newOrg = null;
    organizations = new Vector();
    newOrg = new '+e.org.name+'();
    newOrg.setAgentRole('\'+e.role.name+'\');
    organizations.add(newOrg);
  ),
operationPlays : uml!Operation (
  isAbstract <- false,
  specification <- 'void createPlays()'
)
)

```

3.2.3 The creation of the plans and actions classes

Each agent defines its set of plans and their correspondent actions. In the MAS-ML structural diagrams, the plans and actions are named and associated with the agent. The execution of plans and actions are therefore modeled in MAS-ML dynamic diagrams.

In order to create the correspondent plans and actions by using ASF, it is necessary to create the classes that will represent these agent properties by extending the abstract ASF classes *Plan* and *Action*. Those classes receive the name of the plans and actions defined in the MAS-ML structural diagrams and also the implementation described in the MAS-ML dynamic diagrams. Note that plans are related to the goals that they achieve and to the actions that they execute. Therefore, the constructor method of a plan executes methods to relate the plan instance being created to its goals and actions.

```

rule Plan {
from
  c : masml!Plan
to
outPlan : uml!Class (
  name <- c.name,
  isAbstract <- false,
  feature <- Set{constMethod, createActions, createGoals}
),
constMethod : uml!Method (
  specification <- operationPlan,
  body <- 'createActions(); createGoals()'
),
operationPlan : uml!Operation (
  isAbstract <- false,
  specification <- c.name+'()'
),
-----
-- Relating actions to the plan
createActions : uml!Method (
  specification <- operationActions,
  body <- c.actions -> iterate (e; body : String = '' | body+
    Action newAction = null;
    actions = new Vector();
    newAction = new '+e.name+'();
    actions.add(newAction);
  ),
operationActions : uml!Operation (
  isAbstract <- false,
  specification <- 'void createActions()'
),
)

```



```

-----
-- Relating goals to the plan
createGoals : uml!Method (
    specification <- operationGoals,
    body <- 'Goal newGoal = null;
            newGoal= new Goal('+c.goal.value+', '+c.goal.valueType+', '+c.goal.goalType+');
            goals.add(newGoal);'
),
operationGoals : uml!Operation (
    isAbstract <- false,
    specification <- 'void createGoals()'
))

```

3.2. Applying the Transformations in a Simple Example

The UML model generated by the transformation is a UML class diagram that contains the ASF framework classes and the classes related to the application that instantiate the framework. Since this paper does not concern the MAS-ML dynamic diagrams during the transformation, UML dynamic diagrams are not part of the target model and, therefore, details about the execution of the agents were not transformed. All application entities, properties and relationships modeled on the three MAS-ML structural diagrams are represented in the target UML class model. Figure 3 depicts the transformation of the agent class *UserAgent* modeled in a MAS-ML organization diagram into a set of three classes (and its correspondent methods) modeled in a UML class diagram instantiating ASF.

4 Embracing the Evolution of Applications and Techniques

The evolution of the applications' requirements is easily handled by our MDE approach. After changing the design models according to the updated requirements, the implementation models can be regenerated by using the same set of rules already available. Changes in the requirements of an application do not require adaptations of our approach.

Besides the evolution of applications, the techniques being used in our approach may also evolve. It may occur adaptations in the MAS-ML metamodel, the ASF framework and also in the UML metamodel. These three different technology adaptations influence the transformation rules. The evolution of the MAS-ML and UML metamodel influence the transformation rules since the rules are defined to receive well-formed models according to the source metamodels and to generate well-formed models according to the target metamodel. Adaptation of the ASF framework clearly influence the transformation rules since the generated target models are instances of the framework and, therefore, conforms to its specification. If the framework changes its instances also change.

Since the design and implementation models are low coupling, changes in the MAS-ML metamodel do not influence in the ASF specification or in the UML metamodel, and vice-versa. Besides, the rules were defined to try to minimize the effort of changing the rules due to technique evolution, as detailed below:

- *There is a rule for creating all the classes defined in ASF.* Therefore, if the ASF specification evolves, it will have a minor influence into the whole set of transformation rules. The rule that generates the ASF classes will need to be modified together with few other rules, depending on the adaptation. For instance, if the abstract *Agent* class is modified it may be necessary to modify the *Agent* rule that generates the specializations of the abstract *Agent* class.

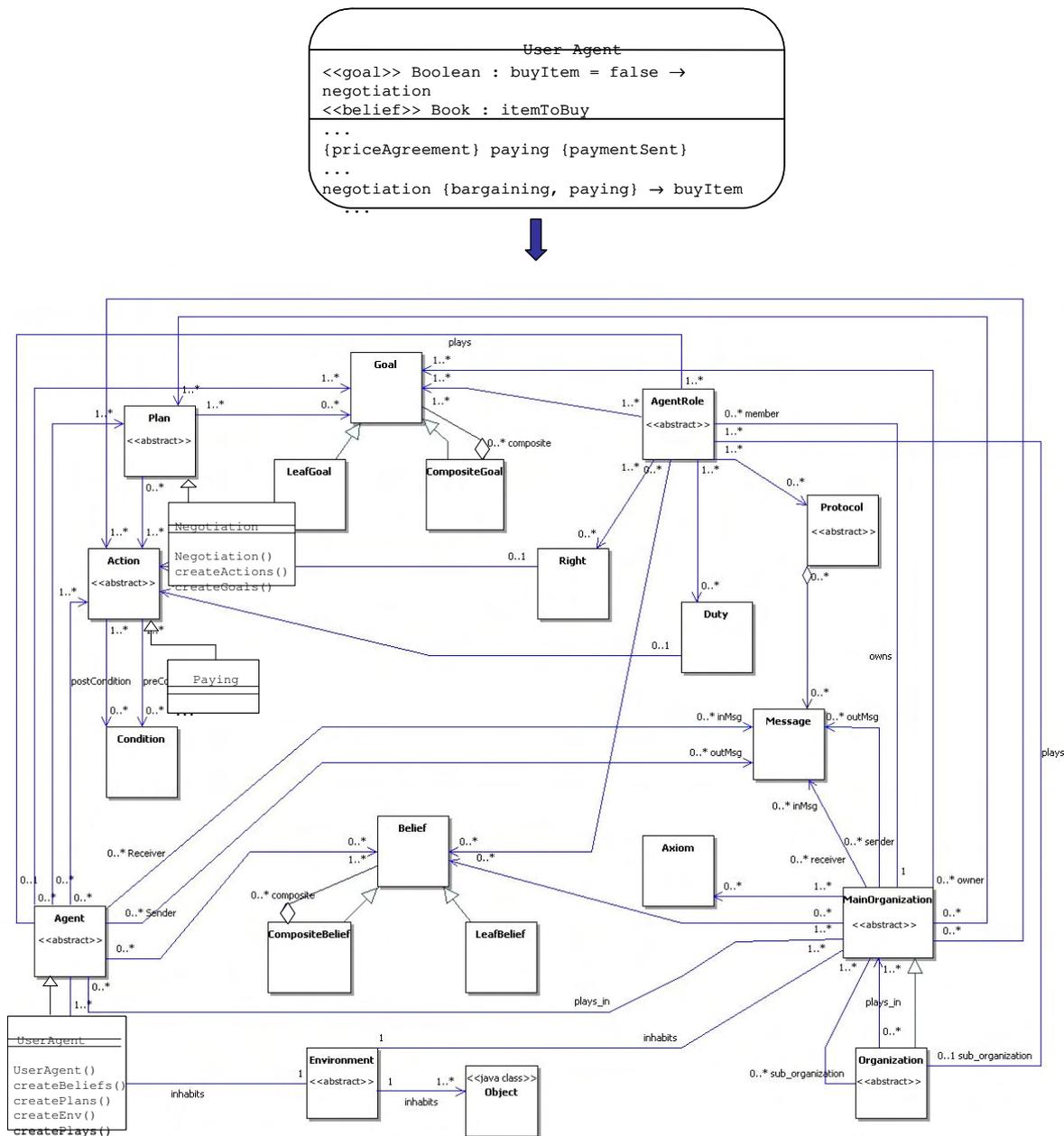


Figure 4. Transformation of a MAS-ML model into a UML model by instantiating ASF

- *There is a rule for each entity type defined in MAS-ML.* If the MAS-ML metamodel evolves, it will be necessary to adapt the correspondent rule(s) that deal(s) with the modified part. Fortunately, each transformation rule deals with the transformation of one entity type defined in the MAS-ML metamodel. For instance, if the part of the metamodel that specifies the entity type *agent* is adapted, it will only be necessary to modify the *Agent* rule.
- *There is a rule for generating the plans, actions and protocols.* Since there is a need for creating OO classes for implementing plans, actions and protocols, we have defined a rule for transforming such properties. Thus, if an adaptation occurs in MAS-ML that



affects how such properties are modeled or in ASF that affects how such properties are implemented, these rules must also be adapted.

- *It is easy to find out the part of the UML metamodel being used in the rules.* Each rule identifies the part of source model that the rule is able to transform and the part of the target model that it can generate. Such identification is done by point out the metaclasses of the source and target metamodels. Therefore, if the UML metamodel is modified, it will be necessary to search in all transformation rules the rules that deal with the modified part. For instance, if the specification of *Method* is changed, it will be necessary to search in all rules the part of the rules that generate methods, i.e., the part of the rules where the sentence *uml!Method* is stated.

5 Related Work

Although, some MAS methodologies such as Prometheus [15], Tropos [2] and MaSE [4] have not used an MDA approach, they have already proposed the mapping between the design models into implementation code and have also provided some tools for supporting both the design and the implementation of MAS. However, they do not clearly demonstrate the mapping from design models into code by presenting the rules used in the transformation. Therefore, it is extremely difficult to use the design models created by using the methodologies to generate code to another platform or framework that has not been addressed by them.

In addition, they do not separate the models into platform independent models and platform specific models. By using some of these methodologies, it is possible to describe platform specific details during the design of the application. In such cases, the high-level design models are platform dependent and, consequently, are not easily portable to any other platform.

Other authors have already used the MDA approach in other to define a MAS development process. Vallecillo et al [21] demonstrate the use of MDA to derive MAS low-level models from MAS high-level models. The authors propose to use the Tropos methodology and the Malaca platform [1] in the MDA approach. Malaca is platform where agents are defined based on the specification and reuse of software components. The high-level models created while using the Tropos methodology are transformed into low-level Malaca models. However, the transformation from the Tropos models into Malaca models is not completely automated. It requires manual intervention. Moreover, such an approach does not deal with the transformation from Malaca models into code.

Novikay [12] analyzes how GR [3] based on the Tropos visual model can be related to MDA. The author interprets the MDA approach as a visual modeling activity where more abstract models are refined in more detailed models, using transformation techniques. This work covers only the requirement stage existent in Tropos. The difference between our approach and this approach is that ours contemplates the PIM, PSM and code stages.

In Kazakov et al. [10], the authors recommended a methodology based on a model-driven approach for the development of distributed mobile agent systems. They define a mobile agent conceptual model for distributed environments and describe a set of components, represented by a collection of intelligent mobile agents. While such an approach focuses on a specific application domain, our approach is a domain-independent development process.



6 Conclusion and Future Work

The MAS development process presented in this paper intends to provide an approach for modeling and implementing MAS by using MDE. We presented a *language translation* approach that is based on the translations between the source metamodel and the target metamodel. We have implemented a set of transformation rules by using the ATL transformation languages and to several MAS such as a supply chain management system [7][8] as well as web-based paper submission and reviewing system [5][23].

The proposed MDE based development process was illustrated by the use of MAS-ML and ASF. Our intention while using such techniques was to demonstrate how complex it is for transforming design models into implementation code due to the use of different paradigms while modeling the applications and while implementing them. Although MAS-ML and ASF are founded in the same agent's properties and characteristics, it is still not an easy task to manually instantiate ASF to implement MAS-ML design models. Therefore, the use of an (semi-)automatic transformer tool that could generate implementation code from design models is especially important while dealing with modeling language and platforms that do not share the same set of properties and characteristics. Although some times it may be very difficult to define transformation rules, once those rules are defined the implementation of any design model can easily be generated.

A prototyping developing tool [6] was created in order to demonstrate the feasibility of our approach. The tool allows the designers to graphically model MAS systems by using MAS-ML and to implement them while generating Java code by using the ASF framework. With the aim of enhancing the tool, several important improvements should be made. First, the transformer that generates code from MAS-ML models should also consider the MAS-ML dynamic diagrams. Second, the tool should make the visualization and also the modification of the UML models that represent the system implementation feasible. In addition, the tool should provide a model checker to analyze and verify the consistency of the different models (MAS-ML models and UML models).

References

- [1] Amor, M.; Fuentes, L.; Troya, J. A Component-Based Approach for Interoperability Across FIPA-Compliant Platforms. Cooperative Information Agents VII, LNAI 2782, p. 266-288. 2003.
- [2] Bresciani, P. Tropos: An Agent-Oriented Software Development Methodology. Int. Journal of Autonomous Agents and Multi-Agents Systems, 8(3):203-236, 2004.
- [3] Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R. and Lowe M. Algebraic Approaches to Graph Transformation. Handbook of Graph Grammars and Computing by Graph Transformation, 1997.
- [4] DeLoach, S. A. Multiagent Systems Engineering: a Methodology and Language for Designing Agent Systems. In: Proc. of Agent Oriented Information Systems, 1999.
- [5] DeLoach, S. A. Analysis and Design using MaSE and agentTool. In Proc. 12th Midwest Artificial Intelligence and Cognitive Science Conference, 2001.
- [6] DeMaria, B.; Silva, V.; Chore, R.; Lucena, C. VisualAgent: A Software Development Environment for Multi-Agent Systems. In: Tool Track of Brazilian Symposium on Software Engineering, 2005.



- [7] Fox, M. S.; Barbuceanu, M., Teigen, R. Agent-oriented Supply-chain Management. *The International Journal of Flexible Manufacturing*, v.12, p.165-188. 2000.
- [8] Huget, M. Agent UML Class Diagrams Revisited. In: *Proc. of Agent Technology and Software Engineering (AgeS)*, 2002.
- [9] Jouault, F, and Kurtev, I. On the Architectural Alignment of ATL and QVT. In: *Proc. of ACM Symposium on Applied Computing, model transformation track*, Dijon, France, 2006.
- [10] Kazakov, M., Abdulrab, H., Debarbouille, G. A Model Driven Approach for Design of Mobile Agent Systems for Concurrent Engineering: MAD4CE Project 2002.
- [11] Kent, S. Model Driven Engineering. In *Proceedings of Third International Conference of Integrated Formal Methods*, Springer, LNCS 2335, pp. 286-298, 2002.
- [12] Novikay, A. Model Driven Architecture approach in Tropos. Technical Report T04-06-03, Istituto Trentino di Cultura, 2004.
- [13] Odell, J., Parunak, H. Bauer, B. Extending UML for Agents. In *Proceedings of Agent-Oriented Information System Workshop at AAAI*, pp. 3-17, 2000.
- [14] OMG, UML: Unified Modeling Language Specification. Version 2.0. Available at: <http://www.omg.org/uml/>. Accessed in: 02/2005.
- [15] Padgham, L, Winikoff, M. Prometheus: A Methodology for Developing Intelligent Agents, In *Proc. of the 1st Int. Joint Conf. on Autonomous Agents and MAS*, 2002.
- [16] Pokahr, A. Braubach, L., Lamerdorf, W. Jadex: Implementing a BDI-Infrastructure for Jade Agents. *Research of Innovation*, 3(3):76-85, 2004.
- [17] Silva, V., Lucena, C. From a Conceptual Framework for Agents and Objects to a Multi-Agent System Modeling Language. *Journal of Autonomous Agents and MAS*, Kluwer, 9(1-2), 2004.
- [18] Silva, V., Cortes, M., Lucena, C. An Object-Oriented Framework for Implementing Agent Societies. Technical Report MCC32/04, PUC-Rio. Rio de Janeiro, Brazil, 2004.
- [19] Silva, V., Choren, R., Lucena, C. Using the MAS-ML to Model a Multi-Agent System. *Software Engineering for Large-Scale Multi-Agent Systems II*, Springer, 2004.
- [20] Sycara, K., Paolucci, M., Van Velsen, M., Criampapa, J. The Retsina MAS Infrastructure. Special joint issue of *Autonomous Agents and MAS*, 7(1-2):29-48, 2003.
- [21] Vallecillo, A., Amor, M., Fuentes, L. Bridging the Gap Between Agent-Oriented Design and Implementation Using MDA. *Autonomous Agents and MAS Workshop*, pp.93-108, 2004.
- [22] Wagner, G. The Agent-Object-Relationship Metamodel. In: *Second International Symposium: From Agent Theory to Agent Implementation*, 2000.
- [23] Zambonelli, F.; Parunak, H. From design to intention: signs of a revolution. In: *Proc. of the 1st Int. Conference on Autonomous Agents and MAS*, pp. 455-456. 2002.



From C++ Refactorings to Graph Transformations

László Vidács* and Martin Gogolla** and Rudolf Ferenc*

*Department of Software Engineering, University of Szeged, Hungary

**Department for Mathematics and Computer Science, University of Bremen, Germany

Abstract. *In this paper, we study a metamodel for the C++ programming language. We work out refactorings on the C++ metamodel and present the essentials as graph transformations. The refactorings are demonstrated in terms of the C++ source code and the C++ target code as well. Graph transformations allow to capture refactoring details on a conceptual and easy to understand, but also very precise level. Using this approach we managed to formalize two major aspects of refactorings: the structural changes and the preconditions.*

Keywords: Metamodel, C++, UML, Graph Transformation, OCL, Refactoring

1 Introduction

The programming language C++ is widely used in industry today. Many applications written in C++ exist which are constantly developed further, for example, to be adapted to modern service-oriented aspects. On the other hand, there is an important current trend in software engineering that focuses on development activities for using models instead of concentrating on code production only.

This contribution tries to narrow the bridge between industrial, code-centric development with C++ and model-centric development employing languages like the Unified Modeling Language (UML). We discuss a C++ metamodel and display first ideas how development and maintenance of C++ artifacts can be performed on instantiations of this C++ metamodel based on refactorings. Our proposal is to express C++ refactorings and development steps as graph transformations. We think it is important to clearly express transformation concepts for an involved domain like C++ software development. Graph transformations possess a sound theoretical basis and allow to express properties on a conceptual level, not only on an implementation level. Surprisingly, graph transformations have not yet been applied for C++ software development.

Graph transformations have been applied for the transformation of metamodels, see for example the work of Gogolla [Gog00] (among many other works on graph transformation on metamodels). In industry, refactoring techniques [Ref06b] are regarded as promising means for software development. Refactoring of C++ code is supported by a variety of tools [Sli06][Ref06a][Xre06]. However, refactorings are



usually considered from the implementation point of view only, not from a conceptual view. A conceptual view on C++ refactorings on the basis of metamodels and graph transformations allows to express properties like refactoring applicability more precise. A conceptual view also opens the possibility for viewing refactorings on the semantical level, for example, in order to describe semantics preserving refactorings or to test whether they are semantics preserving.

The paper is organized as follows. The next section introduces the C++ language metamodel used in this work. Section 3 discusses C++ refactorings on this metamodel in terms of graph transformations. Section 4 gives insight to our implementation. In Section 5 we mention some important contributions of this area. Finally, the paper ends with a short conclusion.

2 C++ Metamodel

Metamodels, which are also called schemas in the re-engineering community, play a very important role in the process of source code analysis. They define the central repository for the whole process, from where the facts can be reached with the help of different transformations.

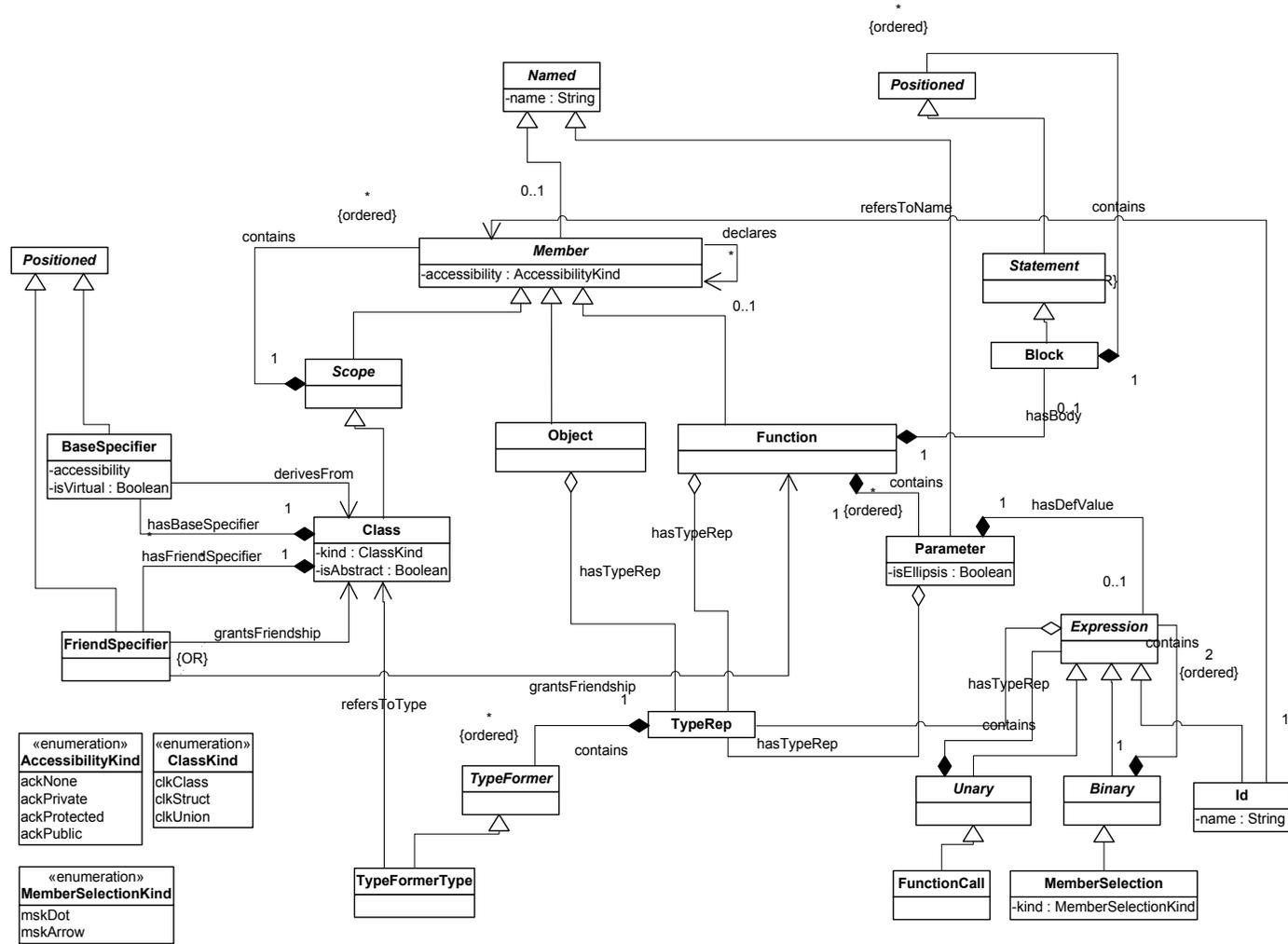
Several researchers have been working on defining metamodels for C++ programs for reverse engineering purposes (also for program comprehension or to define an exchange format) [EKRW02], [Bel00], [FSH⁺01].

The Columbus Schema for C++ [FBTG02] satisfies some important requirements of an exchange format. It reflects the low-level structure of the code, as well as higher level semantic information (e.g., semantics of types). Furthermore, the structure of the metamodel and the used standard notation (UML Class Diagrams) make its implementation straightforward, and what is even more important, an API (Application Programming Interface) is very simple to be realized as well.

Because of the high complexity of the C++ language, the metamodel is divided into six packages. To introduce all packages is beyond the scope of this paper. To clearly present our ideas we created an excerpt mainly from the *struc* package of the metamodel. The presented approach is not limited to this subset of the C++ language, for example templates are also supported by the metamodel. To carry out refactorings on a C++ program it is necessary to deal with preprocessor directives. Although we have a separate metamodel for the preprocessor directives [VBF04], coping with them is not included to this contribution. The excerpt of the C++ metamodel can be seen in Figure 1. The upper part in the figure represents the scoping structure of a C++ program. Class *Member* is the parent of all kinds of elements which may appear in a scope (we use the term “member” in a more general way than usual). In the excerpt there are three important subclasses of *Member*. Class *Class* stands for C++ classes. It may contain further members; it may have base classes (shown by *BaseSpecifier*) and friends. In C++ a friend (class or function shown by *FriendSpecifier* in the figure) can access also protected and private members of the class. The class *Function* stands for C++ functions. It has body (represented by *Block*) which contains any number of *Statements*; and has parameters (class *Parameter*). The class *Object* represents both variables and member fields in a *Class*. In the lower left corner there are the necessary enumerations.

In the middle there are classes for type representation. Like in C, in C++ types can be complex, so each language element which has a type contains a wrapper class called *TypeRep*. A *TypeRep* contains *TypeFormers* (in complex cases a type consists of many code pieces, each piece is a type-former). In the figure many type-formers are omitted, only one is shown (*TypeFormerType*) which refers directly to

Figure 1: Excerpt from the Columbus Schema for C++ - UML class diagram





a type (to a *Class* in this case). This typing structure enables to express all kinds of types and helps to avoid redundancies in storing types. In the lower right corner there are some expressions which are used in this paper (*FunctionCall*, *MemberSelection*, *Id*). An *Id* expression is an identifier in the code which refers to *Member* - in the metamodel this means that it may refer to both classes and class members.

These are the main classes used in our example. For further details please see [Fro06]. As usual in case of complex class diagrams, we cannot express everything using UML class diagram notations easily. For example a function body (block statement) contains ordered *Positioned* nodes. However a *Parameter* is also *Positioned* but it cannot be a part of a function body. OCL expressions as constraints of the class diagram solve many similar issues. We define the following condition (boolean expression, called invariant) that must be true for all *Block* objects:

```
context Block inv:  
not self.contains.ocliIsTypeOf(struc_Parameter)
```

In the following sections we use the introduced metamodel and the OCL expressions together.

3 Graph Transformation Rules on the C++ Metamodel

In this section we show how refactorings on the C++ metamodel can be described with graph transformation. We provide example, which are “classical” refactorings from Fowler’s catalog [Ref06b]. The basic idea is simple in both cases, however many subtle details arise when realizing these refactorings. We also concentrate on C++-specific issues.

3.1 Graph transformation approach, notation

We use a single pushout approach for graph transformation rules possessing a left and a right hand side. Instead of too complicated NACs (Negative Application Condition) we provide preconditions as OCL expressions.

The definition of directed, attributed graphs are used as usual. A program graph is directed, labelled, attributed graph where:

- nodes are labelled with class names shown in the UML class diagram
- nodes have attributes which are called as class attributes, the possible values of the attributes are from the corresponding UML types
- edges are labelled with relation names shown in the UML class diagram

Program graphs are introduced using an object diagram-like notation (see Figure 2).

Not all graphs that correspond to the definition above represent C++ programs. For example an undeclared variable may occur according to the metamodel but the belonging code could not compile. The well formedness is not checked in this paper. In reverse engineering context we assume that the starting graph is a well-formed graph and this property is preserved due to the conditions of transformations. Note that C++ class attributes are modeled with class *Object* in the metamodel - so the nodes representing them have label *Object*. All *Object* and *Function* nodes have *TypeRep* nodes which show their

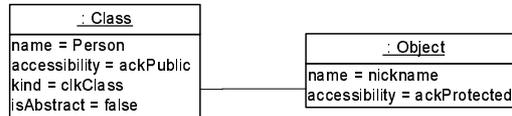


Figure 2: Object diagram like notation of the graph

C++ type. These nodes are omitted from the figures and presented only where it aids comprehension. Furthermore the notation of multi-nodes is introduced to describe general subgraphs. A multi-node with value k represents k pieces of nodes of the same type. Usage of multi-nodes is shown in Figure 3.

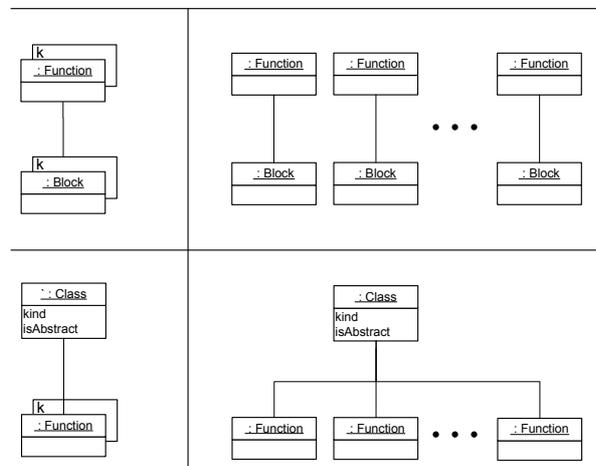


Figure 3: Usage of multi-nodes

3.2 Extract class

Classes should serve a clear, well-defined aim. During development, classes are growing. In lots of cases, there are new responsibilities added to them. The aim of this refactoring is to extract a separate concept and corresponding data to a new class to improve the quality of the design. The idea is shown in Figure 4 which is taken from the Refactoring catalog [Ref06b].

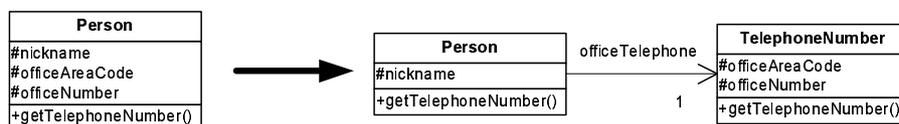


Figure 4: Extract class refactoring example

There is a long way from this semi-informal description to an applicable transformation. Based on this figure we make decisions about the context and purpose of the refactoring. After that we formalize it as graph transformation in two steps: the first part concentrates on the structure of the rule and the second



part declares conditions using OCL expressions.

The main questions to be considered when realizing this refactoring as a graph transformation is: who can use the new (extracted) class and how can it be used. At first, the old class must somehow access it. If it is the only class that uses the new class then their relationship can be implemented either as the new class is a member in the old class or as a dynamic object creation in the constructor of the old class and as a deletion of the object in the destructor of the old class (in this case the other classes can access the new class through public interface functions of the old class). On the other hand, if the new class is free to be used by other classes then it is more complicated - for example a reference counter can be used. This is related to the question: who can instantiate the new class. Similarly, it has to be determined how the new class can be accessed by other classes: through the old class only or through public functions of the new class as well. Another obstacle is introduced by attributes which are used externally from the old class and are now moved to the new class. We assume that the visibility enables one to access those members through a known interface only (this is not a constraint, using the C++ metamodel, all usages of an attribute can be checked).

In this paper we choose to protect the newly created class. The new class can be instantiated only by the old class. Its properties can be modified by the old class, furthermore the old class provides the public interface to use the new class.

As refactorings can be realized in many different ways so we formalize the extract class first as a rule schema. A rule schema has parameters and multi nodes and can be instantiated in concrete cases. We call rule only these concrete cases when the rule schema has concrete arguments and can be applied directly on a program graph. The attributes and operations to be moved must be determined by analyzing their usage. Therefore these are parameters of the transformation: any number of `Object` and `Function` nodes which are contained by the old class. The graph transformation rule schema of `ExtractClass(... : (Object—Function))` is shown in Figure 5.

On the left hand side there is the old class. Its members are divided into 3 groups: attributes, public (interface) functions and protected functions. On the right hand side there are both the old class and the new (extracted) class. The selected attributes and protected functions are completely moved to the new class. Public functions are copied from the old class to the new class. The existing implementation of these functions goes to the copy in the new class. The remaining functions in the old class have a new implementation: they only have to call the copied functions in the new class. The new class cannot be accessed from outside, so the copied public functions became protected.

To ensure the connection between the two classes we have a new member (*Object*) in the old class, its type is the new class. This is represented by new *TypeRep* and *TypeFormerType* nodes. (This abstraction of types is required to represent complex types in C++ [FBTG02].) Now we have to let the old class access the new one, which has protected members/functions. This is done by giving friendship grant to the old class. This is represented by the *FriendSpecifier* node.

Concrete rule and OCL conditions

The left hand side and right hand side of a concrete rule is given in the figures below. The C++ code before the transformation with the left hand side is shown in Figure 6. The graph is in fact the object model of the code, so it contains a node (nickname) which does not take part in the transformation. The

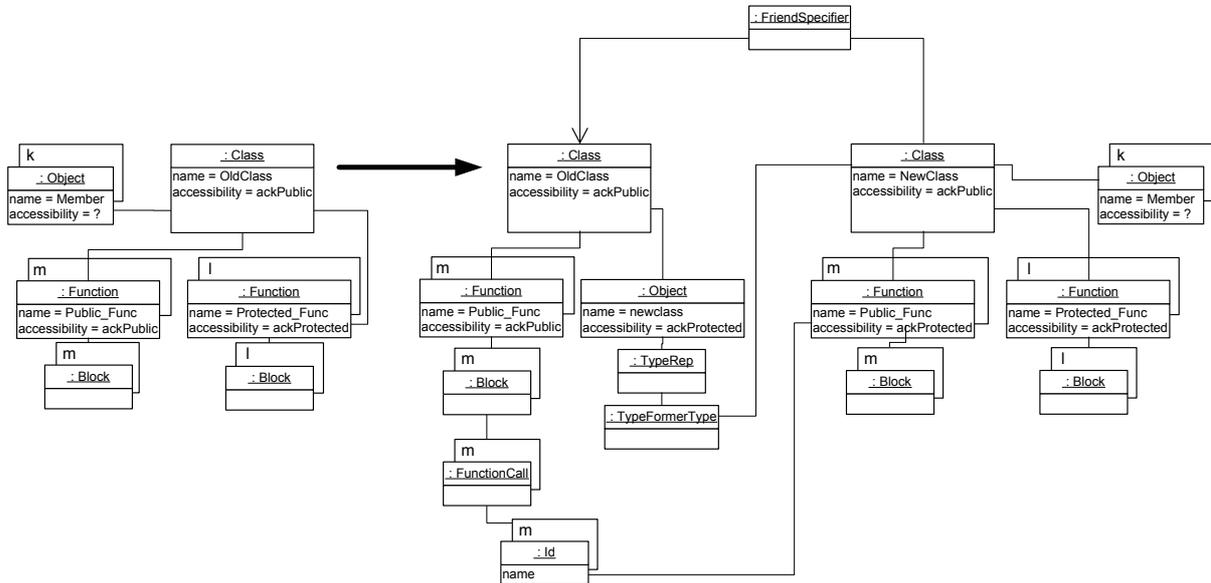


Figure 5: Extract class refactoring as graph transformation

resulting source code and the right hand side is shown in Figure 7.

```
class Person {
protected:
    string nickname;
    string officeAreaCode;
    string officeNumber;
public:
    string getTelephoneNumber ();
};
```

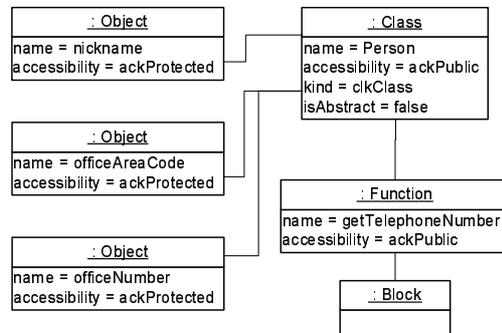


Figure 6: Extract class - C++ code and model instance before the transformation

Moving function from one class to another requires careful examinations of the body. Class members or functions referenced from the body may become unaccessible because of changing the class (they are "foreign" in the new class). (Note that the friendship relation is not symmetric, only the old class can reach the new class.) To prevent accessing unreachable class members we have to check the subgraph of the function body. References to class members/functions are classified based on the relation of the old class and the referenced class as follows: inside the class, class hierarchy (base classes upwards) and outer classes. Bad references that prevent applying the rules are the following ones:

- reference to protected/private member of the old class which is not among the parameters of the



```

class Person {
protected:
    string nickname;
    TelephoneNumber _TelephoneNumber;
public:
    string getTelephoneNumber();
};
class TelephoneNumber {
protected:
    TelephoneNumber();
    string officeAreaCode;
    string officeNumber;
    string getTelephoneNumber();
    friend class Person;
};

```

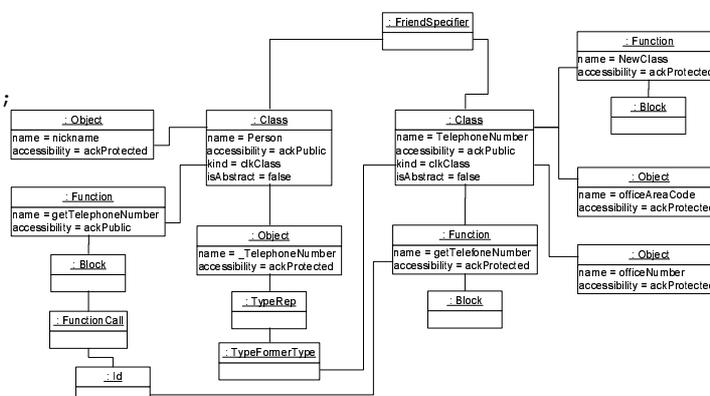


Figure 7: Extract class - transformed C++ code and model instance

rule (if the referenced member is public it means that the extract class refactoring is not so reasonable here)

- reference to protected/private member of one of the base classes (in public case the above note applies)
- reference to protected/private member of an outer class which is a friend of the old class

According to the metamodel a reference is an `Id` node which has a `refersToName` relation to a `Member`, especially to a `Function` or `Object`. The referenced `Member` is contained by a class which is the class we are looking for. To distinguish the 3 different cases mentioned above it is not enough to search `Ids` in the subgraph in special container nodes like `MemberSelection` or `FunctionCall`. It may happen that a member selection contains this pointer and the function call without a `membersselection` may reference a function in a base class. So we have to scan all `Ids` and find the referenced class members/functions and their container classes. Note that in the case of `MemberSelection` expressions the container class can be found through the left hand side child `Id` as well.

The scan can be implemented as an OCL expression.

```

let Old : struc_Class = B1.hasBody.containsMember.oclAsType(struc_Class)
in
let Bases : Bag(struc_Class) = Old.hasBaseSpecifier.derivesFrom
in
let M : Bag(struc_Member) = B1.containsPositioned->select(i | i.oclIsTypeOf(expr_Id)).
    oclAsType(expr_Id).refersToMember.select(oclIsTypeOf(struc_Object) or
    oclIsTypeOf(struc_Function))
in
M.iterate(m : struc_Member; res : Boolean = true |
let Cont : struc_Class = m.contains.oclAsType(struc_Class)
in
--condition A : referenced members are in the old class but they are not
--                among the parameters of the transformation rule

```



```

if ((m.accessibility='protected') or (m.accessibility='private') and
    (Cont=Old) and not (Bag{O2,O3,F1}.exists(i | i=m)) )
then
res and false
else
--condition B : referenced members are in the base classes
if ((m.accessibility='protected') or (m.accessibility='private') and
    (Bases.exists(bc | bc=Old)))
then
res and false
else
--condition C : referenced members are outer friends
let Friends : Bag(struc_Class) = Old.hasFriendSpecifier.grantsFriendship
in
if ((m.accessibility='protected') or (m.accessibility='private') and
    (Friends.exists(fc | fc=Cont)))
then
res and false
else
res
endif
endif
endif
endif

```

The expression checks for wrong references in a function body B1 (argument of the expression). In case of any occurrence of a wrong reference the expression returns false and prevents the rule to be applied. This example show the expressiveness of the OCL. OCL expressions may contain searches through collections which cannot be easily formulated with simple NACs. A general subgraph notation is needed for instance to check whether a class is one of the base classes of the old class.

3.3 Other refactorings

There are several refactorings which can be implemented on our metamodel as graph transformations in a similar way. In this paper we give only the (detailed) case study of extract class not only because of space limitation. It contains many structural changes and also complex preconditions. In a work of Eetvelde [VJ05] there is a list of 15 formalized refactorings in 29 pages. Papers usually demonstrate two refactorings like pull up method and encapsulate variable [MVDJ05], or extract code and move method refactorings [BPPT04]. We may say that presenting more of the above examples will not say more than our extract class example regarding the two important aspects of refactorings: formalization of the structural changes and the preconditions; which was the aim of the current contribution.

4 Implementation

The extract class refactoring graph transformation was implemented using the USE (UML-based Specification Environment) software [USE06] instead of an existing graph transformation engine. USE is a system for the specification of information systems. It is based on a subset of the Unified Modeling Language (UML). A USE specification contains a textual description of a model using features found in



The result of the extract class refactoring in USE can be seen in Figure 8. The execution of the script was quick because the parameters determined the place of the transformation so the modifications were made locally (below 1 sec). Future work is to try this implementation on real life software systems. Running time of the rule-creator script (which creates basic graph operations from a rule) depends on the size of the rule. In general the most time consuming part is identification of the places where the transformation is applicable. In case of refactorings in most cases the programmer has to consider and choose a place to apply the refactoring. The decision is made based on criteria which are not easy to formalize. For instance extract class refactoring may be applicable on almost every class (which has members or functions) but only in few cases it is useful to apply. Thus in a real life software the identification of the big class can be done using other visualization or analyzer tools. After we have identified the place (parameters of the rule), the actual refactoring can be applied quickly - like in our example.

5 Related work

Since the pioneering work of Opdyke [Opd92] there were lots of efforts made to give a formalism for refactorings. Graph transformations are also considered as a basis of formalizing refactorings. Our work has close connections to such approaches. A solid contribution that shows the current state of the art is given in the work of Mens at al. [MVDJ05]. The graph representation of a program plays an essential role in the formalism. The paper describes a language independent formalism and also introduces two major issues in detail: preconditions and behaviour preservation. We borrowed ideas of the graph formalization from Eetvelde at al. [VJ05] like the multi-nodes and edges. Bottoni [BPPT04] uses a similar formalism, the focus in that work is on the coordination of a change in different model views of the code using distributed graph transformations. These works however are not specialized towards C++ as our approach is.

Although there is much progress in this area, industry uses more or less the same solutions as before: language specific refactorings are implemented separately. Fanta and Rajlich [FR98] contribute a natural way of implementing refactorings. The paper shows the key points but this solution is somehow “out of control” without a formal base. They state that these transformations are surprisingly complex and hard to implement. Two reasons they give for that are the nature of object-oriented principles and the language specific issues. We agree with this view, our work shows how to deal with C++ specific issues on meta level with the checking possibilities provided by the OCL. Our work also differs from the others in that instead of the usual graph transformation engines we used a script based OCL solution of the USE system.

6 Conclusion

This paper presents work in progress on using graph transformations to express C++ refactorings in a clear way and on a conceptual level. Although graph transformations have been used for metamodel transformation of various languages, they have not been extensively used for C++. For a successful use of graph transformations, it is necessary to work on cumbersome subjects like C++ refactorings and the accompanying nasty details. We are aware of the fact that many of our concepts are already known from other successful application areas of graph transformation.



Future work will elaborate further C++ language refactorings. We will also improve these refactorings in a graph rewriting machine resp. modeling tool. We think that such an implementation will give insight into the application conditions and properties of C++ refactorings on a conceptual level. Such an implementation will thus enable a deeper understanding of C++ refactorings.²

References

- [Bel00] Bell Canada Inc., Montréal, Canada. *DATRIX – Abstract semantic graph reference manual*, version 1.2 edition, January 2000.
- [BPPT04] Paolo Bottoni, Francesco Parisi-Presicce, and Gabriele Taentzer. Specifying integrated refactoring with distributed graph transformations. *Lecture Notes in Computer Science*, 3062:220–235, 2004.
- [EKRW02] Juergen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. GUPRO - Generic Understanding of Programs. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.
- [FBTG02] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus - reverse engineering tool and schema for C++. In *ICSM 2002*, pages 172–181, Montreal, Canada, October 2002. IEEE Computer Society.
- [FR98] Richard Fanta and Vaclav Rajlich. Reengineering object-oriented code. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 238, Washington, DC, USA, 1998. IEEE Computer Society.
- [Fro06] Homepage of FrontEndART Ltd. <http://www.frontendart.com>, 2006.
- [FSH⁺01] Rudolf Ferenc, Susan Elliott Sim, Richard C Holt, Rainer Koschke, and Tibor Gyimóthy. Towards a Standard Schema for C/C++. In *WCRE 2001*, pages 49–58. IEEE Computer Society, October 2001.
- [Gog00] Martin Gogolla. Graph Transformations on the UML Metamodel. In *GVMT'2000*, pages 359–371. Carleton Scientific, Waterloo, Ontario, Canada, 2000.
- [MVDJ05] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution: Research and Practice*, 17:247–276, 2005.
- [Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.
- [Ref06a] Homepage of Ref++. <http://www.refpp.com>, 2006.
- [Ref06b] Refactoring catalog. <http://www.refactoring.com/catalog/>, 2006.
- [Sli06] Homepage of Slickedit. <http://www.slickedit.com/>, 2006.
- [USE06] Homepage of USE. <http://www.db.informatik.uni-bremen.de/projects/USE/>, 2006.
- [VBF04] László Vidács, Árpád Beszédes, and Rudolf Ferenc. Columbus Schema for C/C++ Preprocessing. In *CSMR*, pages 75–84. IEEE Computer Society, March 2004.
- [VJ05] Niels Van Eetvelde and Dirk Janssens. Refactorings as graph transformations. Technical report, University of Antwerp, February 2005. UA WIS/INF 2005/04.
- [Xre06] Homepage of Xrefactory. <http://xref-tech.com>, 2006.

²László Vidács acknowledges the financial support provided through the European Community's Human Potential Programme under contract HPRN-CT-2002-00275, SegraVis.



Appendix

A USE description

```

-----
-- Extract Class refactoring
-----
rule ExClass
left
  C1:struc_Class
  O1:struc_Object
  O2:struc_Object
  F1:struc_Function
  (C1,O1): containsMember
  (C1,O2): containsMember
  (C1,F1): containsMember
right
  C1:struc_Class
  C2:struc_Class
  O1:struc_Object
  O2:struc_Object
  NO1:struc_Object
  F1:struc_Function
  NF1:struc_Function
  NF2:struc_Function
  NB1:statm_Block
  FS:struc_FriendSpecifier
  TR:type_TypeRep
  TF:type_TypeFormerType
  (C2,O1): containsMember
  (C2,O2): containsMember
  (C2,F1): containsMember
  (C2,NF1): containsMember
  (C1,NF2): containsMember
  (NF2,NB1): hasBody
  (C1,NO1): containsMember
  (NO1,TR): objectHasTypeRep
  (TR,TF): containsTypeFormer
  (TF,C2): refersToClass
  (C2,FS): hasFriendSpecifier
  (FS,C1): grantsFriendshipToClass
-- postconditions
[C2.name = 'TelephoneNumber']
[C2.accessibility = 'ackProtected']
[C2.kind = 'clkClass']
[C2.isAbstract = false]
[C2.isDefined = true]
[NF1.name = 'TelephoneNumber']
[NF1.accessibility = 'ackProtected']
[F1.accessibility = 'ackProtected']
[NO1.name = 'TelephoneNumber']
[NO1.accessibility = 'ackProtected']

```



```

[NF2.name = 'getTelephoneNumber']
[NF2.accessibility = 'ackPublic']
end

-----
-- Extract Class refactoring
-- ExClassGT_ExClass.cmd
-- ExClass(_C1,_O1,_O2,_F1)
-- the 'match' parameter can be bound with '!let match = ...'
-----

!let _C1 = match->at(1)
!let _O1 = match->at(2)
!let _O2 = match->at(3)
!let _F1 = match->at(4)
!openter rc ExClass(_C1,_O1,_O2,_F1)

!create _C2 : struc_Class
!create _NO1 : struc_Object
!create _NF1 : struc_Function
!create _NF2 : struc_Function
!create _NB1 : statm_Block
!create _FS : struc_FriendSpecifier
!create _TR : type_TypeRep
!create _TF : type_TypeFormerType
!insert(_C2,_O1) into containsMember
!insert(_C2,_O2) into containsMember
!insert(_C2,_F1) into containsMember
!insert(_C2,_NF1) into containsMember
!insert(_C1,_NF2) into containsMember
!insert(_NF2,_NB1) into hasBody
!insert(_C1,_NO1) into containsMember
!insert(_NO1,_TR) into objectHasTypeRep
!insert(_TR,_TF) into containsTypeFormer
!insert(_TF,_C2) into refersToClass
!insert(_C2,_FS) into hasFriendSpecifier
!insert(_FS,_C1) into grantsFriendshipToClass
!set _C2.name := 'TelephoneNumber'
!set _C2.accessibility := 'ackProtected'
!set _C2.kind := 'clkClass'
!set _C2.isAbstract := false
!set _C2.isDefined := true
!set _NF1.name := 'TelephoneNumber'
!set _NF1.accessibility := 'ackProtected'
!set _F1.accessibility := 'ackProtected'
!set _NO1.name := 'TelephoneNumber'
!set _NO1.accessibility := 'ackProtected'
!set _NF2.name := 'getTelephoneNumber'
!set _NF2.accessibility := 'ackPublic'
!delete(_C1,_O1) from containsMember
!delete(_C1,_O2) from containsMember
!delete(_C1,_F1) from containsMember
!opexit

```

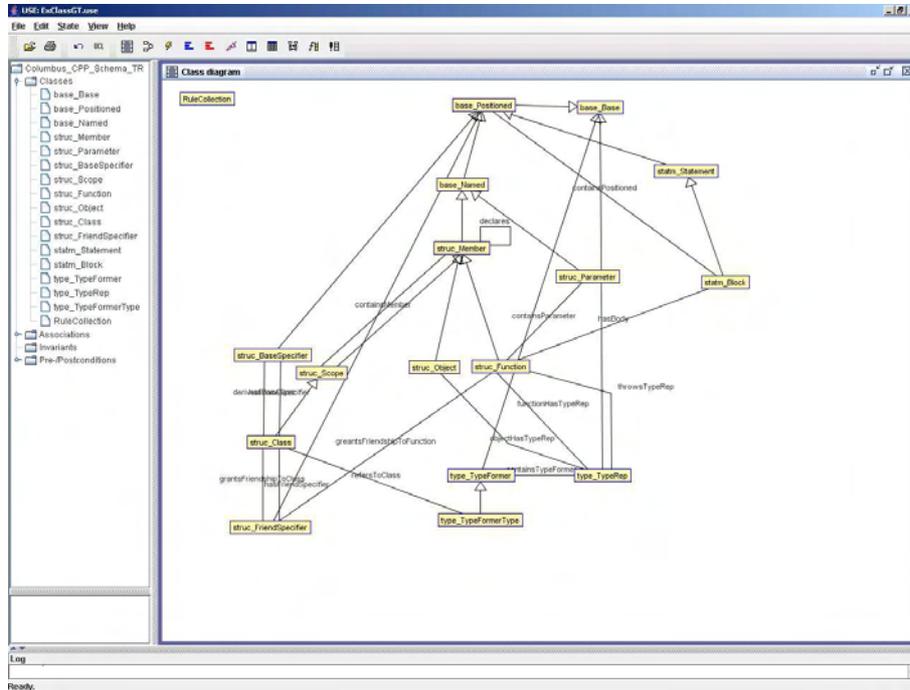


Figure 9: Class diagram of the C++ metamodel excerpt in USE

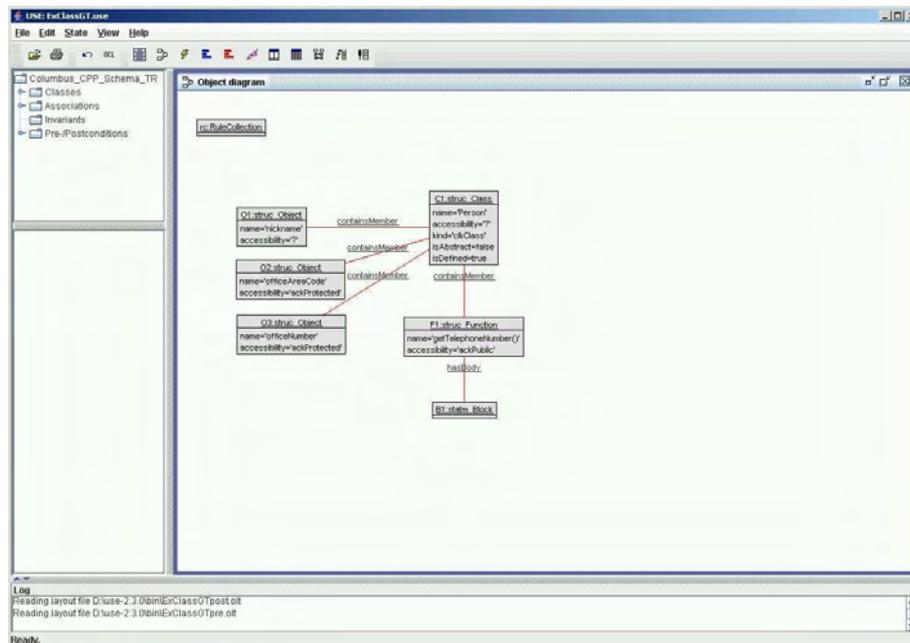


Figure 10: Object diagram in USE before the refactoring