Preliminary Proceedings

International Workshop on Models and Evolution

ME 2011

ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems

Wellington, New Zealand
October 16-21, 2011

http://www.di.univaq.it/ME2011/

Table of Contents

A Domain Specific Transformation Language

Bernhard Rumpe and Ingo Weisemöller

Software Engineering
RWTH Aachen University, Germany
http://www.se-rwth.de/

Abstract. Domain specific languages (DSLs) allow domain experts to model parts of the system under development in a problem-oriented notation that is well-known in the respective domain. The introduction of a DSL is often accompanied the desire to transform its instances. Although the modeling language is domain specific, the transformation language used to describe modifications, such as model evolution or refactoring operations, on the underlying model, usually is a rather domain independent language nowadays.
Most transformation languages use a generic notation of model patterns that is closely related to typed and attributed graphs or to object diagrams (the abstract syntax). A notation that reflects the transformed elements of the original DSL in its own concrete syntax would be strongly preferable, because it would be more comprehensible and easier to learn for domain experts. In this paper we present a transformation language that reuses the concrete syntax of a textual modeling language for hierarchical automata, which allows domain experts[1] to describe models as well as modifications of models in a convenient, yet precise manner. As an outlook, we illustrate a scenario where we generate transformation languages from existing textual languages.
Keywords: domain specific languages, model transformations.

1 Introduction and Problem Statement

Domain specific languages (DSLs) have the advantage of allowing domain experts to model parts of the system in a problem-oriented notation that is well-known in the respective domain. Like most documents in software development processes, models in DSLs underly frequent changes. These may include refactorings, automated modifications, or complex editing operations. Change operations on models can be described in explicitly defined model transformations.

To define a model transformation, we need an appropriate transformation language. Today's transformation languages [7] however operate on the abstract syntax and thus look very different from the DSL to be transformed. In the following sections, we are going to present an approach to close this gap.

[1] In our wording, the term "domain" refers to application domains such as business processes or a discipline of engineering as well as to technical domains such as relational databases or state based systems.

If the user wants to keep the look-and-feel of the DSL within the transformation language, then this language needs to embody elements of the concrete syntax of the underlying DSL, and is thus domain specific itself. Consequently, instead of having a single language for transformations of models in arbitrary DSLs, we would prefer a syntactically fitting transformation language that provides the same look-and-feel as the DSL at hand.

In this contribution, we state that the concrete syntax of a textual DSL can be reused to describe transformation rules, thus providing this look-and-feel. We substantiate our claim by the introduction of a transformation rule used in the process of flattening hierarchical automata and of the corresponding transformation language. Because the elements of the transformation language depend on the elements of the automata language in a systematic manner, we believe it is possible to systematically if not automatically derive the transformation language from a given DSL.

The following sections are outlined as follows: In Section 2 we provide a brief introduction to graph based model transformations, based on a rule used in the process of flattening hierarchical automata. We are going to reuse this example in the subsequent sections. Section 3 gives an introduction to existing approaches to the definition of model transformations in a domain specific notation. In Section 4 we explain what transformation rules in concrete syntax look like. In Section 5 we summarize the previous sections and give an overview of our ongoing and future work in this area.

2 Abstract and Concrete Syntax in Transformations

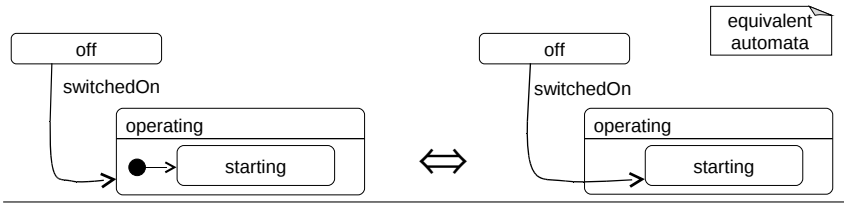
In the following, we consider *transformation rules* to be small steps of transformation in an appropriate language, which may be composed to more complex transformation sequences by control structures or rule application strategies. Composition mechanisms may vary (cf. [7, 17]), whereas we encounter some kind of transformation rules in almost any transformation language. Therefore and for reasons of space, we leave composition mechanisms out of consideration. Instead, we focus on the notation of transformation rules.

In graph based transformation approaches, rules consist of a left hand side (LHS) and a right hand side (RHS), which describe excerpts from a model that the transformation can be applied to (see [15, 12]). Informally explained, the LHS describes a part of the model before the application of the transformation rule, whereas the RHS describes the same part of the model after the rule application.

Because we basically describe excerpts from models, i.e. instances of a modeling language, in the LHS and RHS of a transformation rule, it seems natural to reuse the syntax of the modeling language when describing transformation rules. In current transformation approaches however, this reuse is limited to the abstract syntax for a variety of reasons, which means that the concrete syntax of the modeling language is not reflected in the transformation language.

We are going to show the difference between reusing the abstract syntax only and reusing both abstract and concrete syntax based on a transformation

rule for hierarchical automata. The rule we consider is used in the process of flattening hierarchical automata, which is a simplified case of the transformations for flattening UML state machines (cf. [18, pp. 227 ff.] for details).



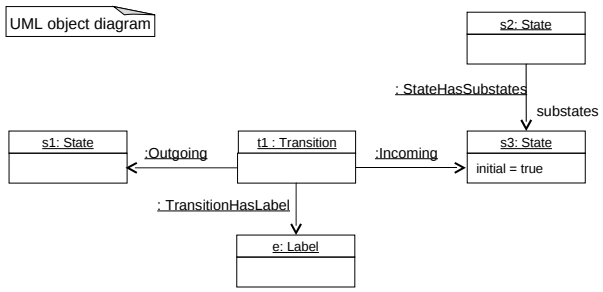

```
1  state off;                                    state off;
2
3  state operating {                             state operating {
4    state starting <<initial>>;      <=>          state starting;
5  }                                             }
6
7  off -switchedOn> operating;                   off -switchedOn> starting;
```

Fig. 1. Equivalent automata in graphical and textual representation

Before investigating the transformation rule itself, we take a look at the syntax of the DSL for hierarchical automata. Figure 1 shows both a graphical and textual representation of a hierarchical automaton on the left, and a graphical and textual representation of a semantically equivalent automaton on the right. The automaton on the right is obtained from the one on the left by forwarding the transition to the nested initial state. The upper part of the figure shows the automata in a graphical syntax, whereas the notation in the lower half is a textual representation of the same automata.

A model transformation rule that can transform an automaton on the left into the equivalent automaton on the right consists of two parts: a LHS, which matches a part of the automaton similar to the left side of Figure 1, and a RHS, which specifies the replacement, and which is similar to the right side of Figure 1. We ignore the RHS of the rule for the moment and take a look at the pattern matching part on the left only: Figure 2 shows the difference between a pattern based on the abstract syntax of the textual DSL from Figure 1, and the same pattern in a notation based on the concrete syntax of that language.

The language of the object diagram pattern in the upper part of the figure reuses the abstract syntax of the automata DSL. The same applies to the second notation (which we did not define explicitly, but is inspired by MOF QVT [11] and OCL [10]). Please note that these patterns are written in pseudocode rather than being executable by some tool, but they depict the general style of transformation languages based on the abstract syntax.



Automaton pattern in OCL-like abstract syntax

```
1  s1 : State;
2  s2 : State;
3  s3 : State;
4  t : Transition;
5  e : Label;
6
7  s2.substates->contains(s3);
8  s3.initial = true;
9  t.source = s1;
10 t.target = s2;
11 t.label = e;
```

Automaton pattern in concrete syntax

```
1  state $source;
2
3  state $outer {
4    state $inner <<initial>>;
5  }
6
7  $source -$event> $outer;
```

Fig. 2. Three variants of the LHS of a rule for transition forwarding

The statements in this pattern are either declarations of typed objects (ll. 1-5), links (l. 7) or additional constraints for these objects (ll. 8-11).

In comparison to this, the pattern based on the concrete syntax of the DSL, which is shown in the lower part of Figure 2, is more compact and easier to read. This is because the transformation language used here is close to the underlying modeling language rather than based on lists of objects, links and constraints.

The pattern matching language differs from the modeling language itself mainly in the use of *schema variables* such as `$source` that act as placeholders for concrete elements from the model.

3 Related Work

A number of publications addresses the specification of model transformations in a notation close to or identical to the corresponding modeling languages. This is usually referred to as *transformations in concrete syntax*, for instance by T. Baar and J. Whittle [3] as well as by M. Schmidt [19]. We will adopt this term for the remainder of this paper.

Both publications mentioned above adapt the concrete syntax of visual modeling languages for the specification of transformation rules. In either approach, the adaption of the syntax is performed manually. To our best knowledge, there is no implementation of either of these approaches available.

Several researchers have spent work on the derivation or inference of transformation rules from concrete examples [13, 22, 5]. In comparison to our work, these approaches usually require a manual adaption of the inferred rule in order to make it applicable to more models than the example it was derived from.

There are also approaches to define a specific transformation language manually, such as JTL [6] or the language for Java patterns presented in [2]. The manual definition of comparable languages for DSLs would be very tedious. Therefore, the generation of transformation languages that we present as an outlook in Section 5 can substantially save efforts for language developers.

Existing approaches to the generation of transformation languages that reflect the concrete syntax of the transformed models are currently limited to graphical, metamodel-based languages. The most mature approaches we are currently aware of are the ones by R. Grønmo [8] and by Kühne et al. [14]. These approaches however do not consider the concrete syntax of textual languages defined in grammars, and the generation of the languages is not fully automated.

Models in textual languages can also be transformed by term rewriting systems. E. Visser presents how term replacements can be written using the concrete syntax of the underlying language in [23]. Term rewriting rules however are usually limited to a connected (and typically small) subgraph of the target syntax tree, whereas in model transformations we often have to deal with rules that operate on objects distributed all over the syntax tree or even different input files.

Another example where the same language is used to describe expressions and transformations is mathematics and maybe proof systems close to mathematics [4, 16]. Mathematical equations can be understood as transformations, and indeed the success of mathematics to a large extent comes from a precisely defined, composable set of transformation rules (equalities) that allow to manipulate and simplify mathematical formulas in almost any form. Mathematics however does not need explicit references to the abstract syntax.

In conclusion, model transformations for textual languages could be substantially improved in terms of reflecting the concrete syntax of the underlying DSL, and — as far as can be seen from existing work — such transformation languages can to a wide extent be generated from the original modeling languages.

4 Syntactic Form of Transformation Rules

We now introduce the transformation rule language for hierarchical automata. This introduction is informal in the sense that it points out the style of transformation rules and what happens at execution time of these rules. We are not going to completely define their syntax and their semantics, but concentrate on the presentation style of transformation rules. For the understanding of this section, we assume that the reader has a basic knowledge of model transformations, especially model transformations based on graph transformations, and the application of transformation rules to host models as discussed in [1].

We pick up the example of forwarding transitions in automata to nested initial states (cf. Section 2). Figure 3 shows two possible notations of a transformation rule for the forwarding of a single transition, given in concrete syntax[2].

In our approach, which is shown in the upper half of Figure 3, a transformation rule consists of an integrated notation of its LHS and its RHS. In comparison to separate notations, as shown in the lower half of Figure 3, this has two major advantages: The first one is reduced redundancy between the LHS and the RHS, especially if we have a lot of elements that are not changed by the transformation and occur on both sides of the rule. The second one is the possibility to determine object identity between the LHS and the RHS: If an object does not have a name (such as the transition in lines 8 and 16 in the lower part of the figure), or if the name of an object is changed by the transformation rule, we have to introduce additional object IDs, such as `$T` in the example, for defining identical objects on the LHS and RHS.

In our example there are two differences between the LHS and the RHS. In the integrated notation, differences are indicated by a replacement inside the rule, denoted between square brackets `[[...]]` and the replacement operator `:-`. The first difference is defined in line 6: The state identified by `$inner` is not initial on the RHS, indicated by the removal of the modifier `<<[[initial :- ]]>>`. The second one occurs in line 9. The name of the target state of the transition modeled here is `$outer` on the LHS, but `$inner` on the RHS.

These differences describe exactly the modifications that are necessary to transform the LHS from our initial example (cf. Figure 1) into the RHS, i.e. into the automaton that has no nested initial states.

In contrast to our initial example from Figure 1, we do not have to use concrete identifiers of states or transition labels in the transformation rule. Instead of identifiers, we can use schema variables. In our transformation rule language, identifiers are interpreted as schema variables if and only if they start with a dollar sign. Thus, we can unambiguously distinguish between schema variables that must be matched when the transformation is executed, and fixed identifiers. Please note that schema variables cannot only be matched against identfiers, but against arbitrary syntactical elements. For example, `$event` in Figure 3 could also be matched against complex labels with events and preconditions.

[2] This is a simplified rule; actually a semantics preserving transformation would have to forward all incoming transitions to all nested initial states.

Transformation rule in concrete syntax

```
1 state $source;
2
3 state $outer {
4   state $inner << [[ initial :- ]] >>;
5 }
6
7 $source -$event> [[$outer :- $inner]];
```

Transformation rule in concrete syntax, separated LHS and RHS

```
1 match {
2   state $source;
3
4   state $outer {
5     state $inner << initial >>;
6   }
7
8   Transition $T [[ $source -$event> $outer; ]]
9 } replace {
10  state $source;
11
12  state $outer {
13    state $inner << initial >>;
14  }
15
16  Transition $T [[ $source -$event> $inner; ]]
17 }
```

Fig. 3. Transformation rule for transition forwarding in concrete syntax with integrated notation of LHS and RHS (top) or separated notation (bottom)

When a transformation is executed, the transformation engine attempts to find a match for the pattern specified on the LHS. A schema variable may occur several times in a pattern, such as `$source` in lines 1 and 7. In this case, all matches against this schema variable in the host model must have the same value[3]. Elements that have a fixed identifier are matched against the element with exactly this identifier in the host model.

Once a match for the LHS is found, it is replaced by the RHS of the transformation rule. As our work does not focus on the definition of another transformation engine, but we need such an engine to demonstrate our approach, we

[3] This part of the matching is currently limited to a flat, global namespace. We are going to to apply this concept to more complex namespaces in future work.

have chosen a fairly standard way of interpreting the transformation as inspired by graph grammar tools like [21, 20, 1] in the first attempt.

The graph matching approach allows (but not enforces) a match to have properties that are not given in the rule. For example, the initial state given in line 4 of the example may be mapped to a state that is both initial and final.

Our language also provides mechanisms to combine concrete syntax patterns with abstract syntax, thus allowing to define objects with abstract types or additional constraints referring to the abstract syntax. Moreover, it includes advanced concepts for pattern matching in attributed graphs such as sets of nodes or negative application conditions.

5 Conclusions, Current State and Future Work

In the previous section we gave an example of a transformation rule in concrete syntax, which is an instance of a domain specific transformation language. The systematic derivation of such transformation languages from DSLs as well as further improvements of our transformation engine are subject to our ongoing work in this area.

Our goal is to generate transformation languages from the grammars of DSLs. A configuration of the generation process (such as the specification of the variable prefix described above) would be acceptable, but the development of a transformation language in concrete syntax should not require writing source code in a programming language or modifying grammars manually.

We use the MontiCore tool set and framework for the language definition and all generation processes. MontiCore [9] allows for the integrated definition of the concrete and the abstract syntax of DSLs in a grammar format similar to EBNF. It also provides mechanisms to efficiently process models in these DSLs, for instance static analyses, code generation, or model transformations written in Java. Future versions of MontiCore will also provide support of model transformations as presented in this paper.

Figure 4 depicts the process of modeling and transformation language development as well as the usage of these languages according to our approach. A language developer defines the syntax of a DSL in a context-free grammar. From this definition, the rule language generator can automatically derive a grammar of a transformation language and a matching code generator, where the code generator also includes a language independent runtime environment. A domain expert can now not only define models in the DSL, but also implement model transformations in the generated language, which can be processed by the transformation language parser and code generator.

As a proof of concept we are currently working on more complex transformations, including the complete process of flattening UML state machines by transformations in concrete syntax. In order to develop more complex transformations in a manageable way, we are also working on a control flow language, which is syntactically and semantically close to a subset of Java and includes transformation rules as statements or expressions.

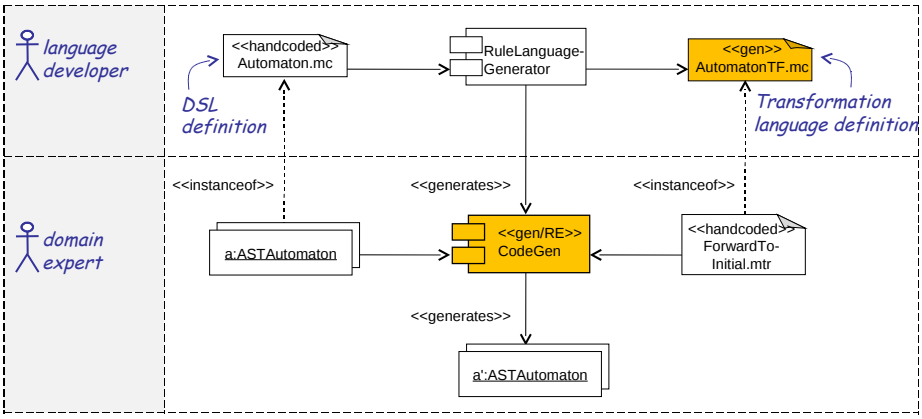


Fig. 4. Roles, documents and components in the transformation process

The current prototype can already execute a subset of the rules required to flatten UML state machines. We plan to come up with a more stable version that includes the control flow language and fully enables the transformations of state machines as well as some other languages in an upcoming version of MontiCore, which we plan to release by the end of this year.

References

1. Agrawal, A., Karsai, G., Shi, F.: A UML-based graph transformation approach for implementing domain-specific model transformations. International Journal on Software and Systems Modeling (2003)
2. Appeltauer, M., Kniesel, G.: Towards concrete syntax patterns for logic-based transformation rules. Electron. Notes Theor. Comput. Sci. 219, 113–132 (November 2008)
3. Baar, T., Whittle, J.: On the Usage of Concrete Syntax in Model Transformation Rules. In: Proc. of Sixth International Andrei Ershov Memorial Conference, Perspectives of System Informatics (PSI). pp. 84–97. Lecture Notes in Computer Science (2006)
4. Bauer, F.L., Ehler, H., Horsch, A., Möller, B., Partsch, H., Paukner, O., Pepper, P.: The Munich Project CIP, Volume II: The Program Transformation System CIP-S, Lecture Notes in Computer Science, vol. 292. Springer (1987)
5. Brosch, P., Langer, P., Seidl, M., Wieland, K., Wimmer, M., Kappel, G., Retschitzegger, W., Schwinger, W.: An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example. In: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems. pp. 271–285. MODELS '09, Springer-Verlag, Berlin, Heidelberg (2009)
6. Cohen, T., Gil, J.Y., Maman, I.: Jtl: the java tools language. SIGPLAN Not. 41, 89–108 (October 2006)
7. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–645 (2006)

8. Grønmo, R.: Using Concrete Syntax in Graph-based Model Transformations. Ph.D. thesis, Dept. of Informatics, University of Oslo (2009)
9. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: Monticore: a framework for the development of textual domain specific languages. In: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume. pp. 925–926 (2008)
10. Object Management Group: Object Constraint Language Version 2.0 (OMG Standard 2006-05-01) (2006), `http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf`
11. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (2008-04-03) (April 2008), http://www.omg.org/spec/QVT/1.0/
12. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Satellite Events at the MoDELS 2005 Conference. Lecture Notes in Computer Science (LNCS), vol. 3844. Springer (2005), http://dx.doi.org/10.1007/11663430_14
13. Kindler, E., Wagner, R.: Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Tech. Rep. tr-ri-07-284, Software Engineering Group, Department of Computer Science, University of Paderborn (June 2007), http://www.uni-paderborn.de/cs/ag-schaefer/Veroeffentlichungen/Quellen/Papers/2007/tr-ri-07-284.pdf
14. Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.: Explicit transformation modeling. In: Ghosh, S. (ed.) Models in Software Engineering, Lecture Notes in Computer Science, vol. 6002, pp. 240–255. Springer Berlin / Heidelberg (2010)
15. Nagl, M.: Graph-Grammatiken: Theorie, Anwendungen, Implementierung. Vieweg (1979)
16. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic. Springer (2002)
17. Philipps, J., Rumpe, B.: Refactoring of Programs and Specifications. In: Kilov, H., Baclawski, K. (eds.) Practical foundations of business and system specifications, pp. 281–297. Kluwer Academic Publishers (2003)
18. Rumpe, B.: Modellierung mit UML. Springer (2004)
19. Schmidt, M.: Transformations of UML 2 Models using Concrete Syntax Patterns. In: RISE 2006 International Workshop on Rapid Integration of Software Engineering techniques. Lecture Notes in Computer Science (LNCS), vol. 4401, pp. 130–143. Springer Verlag, Heidelberg (2006), http://www.springerlink.com/content/836phwk78782v614/
20. Schürr, A.: Progres: A vhl-language based on graph grammars. In: Graph-Grammars and Their Application to Computer Science (LNCS 532). pp. 641–659 (1990)
21. Taentzer, G.: Agg: A graph transformation environment for modeling and validation of software. In: Applications of Graph Transformations with Industrial Relevance. pp. 446–453 (2004), http://dx.doi.org/10.1007/b98116
22. Varró, D.: Model Transformation by Example. In: Proc. 9th International Conference on Model Driven Engineering Languages and Systems (MODELS 2006). LNCS, Springer, Genova (October 2006), http://www.springerlink.com/content/a34jhvh4j01l7514/
23. Visser, E.: Meta-Programming with Concrete Object Syntax. In: Batory, D., Consel, C., Taha, W. (eds.) Generative Programming and Component Engineering (GPCE'02). Lecture Notes in Computer Science, vol. 2487, pp. 299–315. Springer-Verlag, Pittsburgh, PA, USA (October 2002)

Beyond MOF Constraints – Multiple Constraint Set Metamodelling for Lifecycle Management

Keith Duddy keith.duddy@qut.edu.au
Jörg Kiegeland joerg.kiegeland@qut.edu.au

Queensland University of Technology, 2 George St, Brisbane, 4000, Australia,

Abstract. The management of models over time in many domains requires different constraints to apply to some parts of the model as it evolves. Using EMF and its meta-language Ecore, the development of model management code and tools usually relies on the metamodel having some constraints, such as attribute and reference cardinalities and changeability, set in the least constrained way that any model user will require. Stronger versions of these constraints can then be enforced in code, or by attaching additional constraint expressions, and their evaluations engines, to the generated model code. We propose a mechanism that allows for variations to the constraining meta-attributes of metamodels, to allow enforcement of different constraints at different lifecycle stages of a model. We then discuss the implementation choices within EMF to support the validation of a state-specific metamodel on model graphs when changing states, as well as the enforcement of state-specific constraints when executing model change operations.

1 Introduction

The Eclipse Modelling Framework (EMF) is the most commonly used implementation of the Essential MOF specification [Object Management Group, 2008]. Metamodels specified in the EMF Ecore language result in generated code to support instances of a modelled language or domain, in which the instances must strictly conform to constraints inherent in the metamodel, such as cardinalities and attribute changeability. However, the models of many domains require that these constraints change over time, as the objects modelled go through lifecycle stages, which necessitates the specification of metamodels that contain the weakest constraints required at any stage of a model's life cycle. This requires the addition of "business rules" within the supporting repository and tools to enforce tighter constraints than the metamodel does, or constrain the user interface from allowing a user to put a model into certain states at different lifecycle stages.

A related problem that we have encountered is a situation where there is a published *normative* metamodel, which represents some idealised set of constraints that apply to models which are already constructed and in use. In the first instance, there is always a stage of building a model, at which point the instance cannot possibly conform to the metamodel until a set of relevant instances and values are created and linked together. Standard metamodels may be too restrictive to reflect the reality of the domain, and only express the constraints of an idealised state of the model, which may only be achieved after some process. In this case, the tools need to selectively enforce the metamodel constraints, or rely on a a parallel implementation of a more relaxed constraint model, thereby compromising model interoperability.

Our approach to dealing with these situations is to allow for the creation of variations of a metamodel in which the classes, attributes and references are all the same, but their constraining meta-attributes are associated with a named lifecycle stage for instances. We

represent the lifecycle stages as a state-machine specification, with each stage represented by a state.

Using Ecore we represent variations as annotations to the meta-attributes which are labelled with the name of the state in which the variation should be enforced. Two special variations, which may or may not be represented in the state machine are created first: the "normative" and "relaxed" variations. The normative metamodel is usually received from a published standard. The relaxed metamodel is generated using model transformation to alter the meta-attributes under consideration. The relaxed metamodel is then augmented using the Ecore model annotation mechanism to embed the normative meta-attributes values as annotations. Following this, variations for each of the states representing lifecycle stages are then embedded as additional annotations.

We then associate a state machine representing the lifecycle stages with the each "model instance" (actually a graph of instances connected by containment references), and generate code which looks up the current state to determine which constraints to enforce when models are validated or have change operations executed on them.

The next section provides a context for the approach, and describes a metamodel example which motivates it. The details of the implementation are explained in Section 3. Future work is discussed in Section 4, and related work is discussed in Section 5. The conclusion is given in Section 6.

2 Motivating Context and Metamodel Example

The context for this work is in the management of model repositories created from metamodels of data. We have a tool set which generates Web service accessible repositories of data sets from Eore models called *Repository as a Service (RaaS)*. The RaaS tools generate only as many Web service operations in a repository's WSDL interface (or URI mappings in the REST interface) as needed to transfer coherent sub-graphs of objects which are connected through containment references. The design rationale and initial implementation are documented in [Duddy et al., 2010]. Currently the tools are being extended to generate customised interface support for multiple stakeholders, who have different constraints depending on their role and the lifecycle stage of a shared model. This paper simplifies the problem under consideration to a single user over many lifecycle states of the model. We restrict the discussion of constraints for this paper to the enforcement of lower bounds on attribute and reference cardinalities, and the changeability meta-attribute.

The initial use case for repository generation was for storing service descriptions as specified by the MOF metamodel of the Unified Service Description Language (USDL)[1] [Cardoso et al., 2009], which is a large metadata set for describing human- and computer-provided services. The USDL covers several aspects of service description, and includes participant, technical, legal, pricing and service level modules. A more robust implementation of the RaaS tools used to create a USDL repository is described in the [Barros & Oberle, 2011]. The example that will be used in this paper is that of a *Price Plan* for a Service. See Figure 1 for a subset of the metamodel as it appears in the USDL specification. We require differing constraints when the service description transitions between three lifecycle stages: *Preparation for Offer*, *Offered* and *Deprecated*. The state machine showing the transitions allowed between these stages is shown in Figure 2.

Currently we are also considering the interactions of several stakeholders in the building and construction industry through a model repository containing *Building Information*

[1] Language specification can be obtained at http://internet-of-services.com, last accessed 28 July 20111

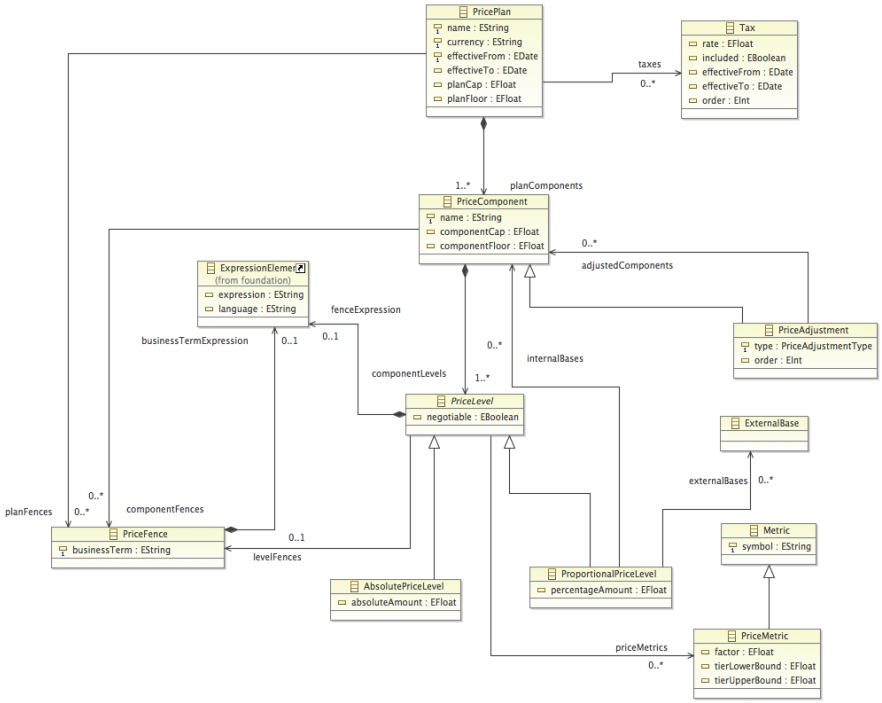


Fig. 1. The Normative USDL Price Plan metamodel

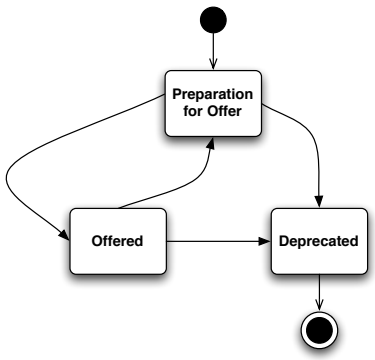


Fig. 2. The Simplifed Lifecycle Stages of a USDL Service Description

Models (BIM) [Succar, 2009], [Steel et al., 2010]. Similar variations to the constraints on BIM models occur over their lifetime, and many of these are best described as variations to the metamodel. The lifecycle is complex, and includes the design, engineering, quantity surveying, construction and maintenance of buildings modelled as BIMs.

The Price module of USDL describes a very general framework for modeling price plans for services. Its detailed design is described in [Kiemes & Oberle, 2010], and in the USDL specifications. In the life cycle of a service description the following requirements on prices are applicable in the three states described by our simplified lifecycle state machine (Figure 2):

Preparation for Offer There must be no lower bounds on any attributes or references. All attributes and references must be changeable.

Offer All lower bounds must be enforced as in the normative metamodel, except that the lower bound for *taxes* must be 1, as all prices are subject to VAT in the marketplace. All references except *taxes* and *adjustedComponents* must not be changeable – which allows taxes to be applied according to legislation, and discounts to be introduced. Only the *name* attributes can change when in this state (although these are also not expected to change). All other attributes must be fixed as they represent the offered service price plan, which is required to be unchanged during the offer period.

Deprecated All lower bounds must be enforced as in the normative metamodel. All attributes and reference must not be changeable.

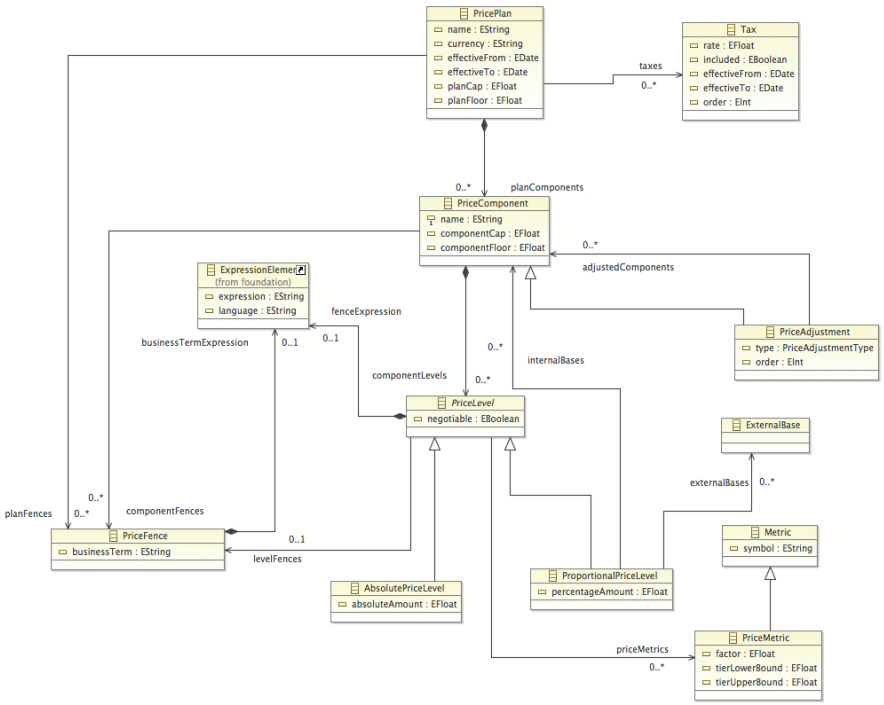


Fig. 3. The Relaxed variation of the Price Plan metamodel

Note that due to a lack of standard graphical representations for the Ecore *changeable* meta-attribute, we use a clear box next to an attribute to represent *true* and a black box to represent *false*. For references we use an open arrow for *true* and a black arrow for *false*. This allows for easy visual differentiation when comparing metamodel variants, as can be seen when comparing Figures 3 and 4.

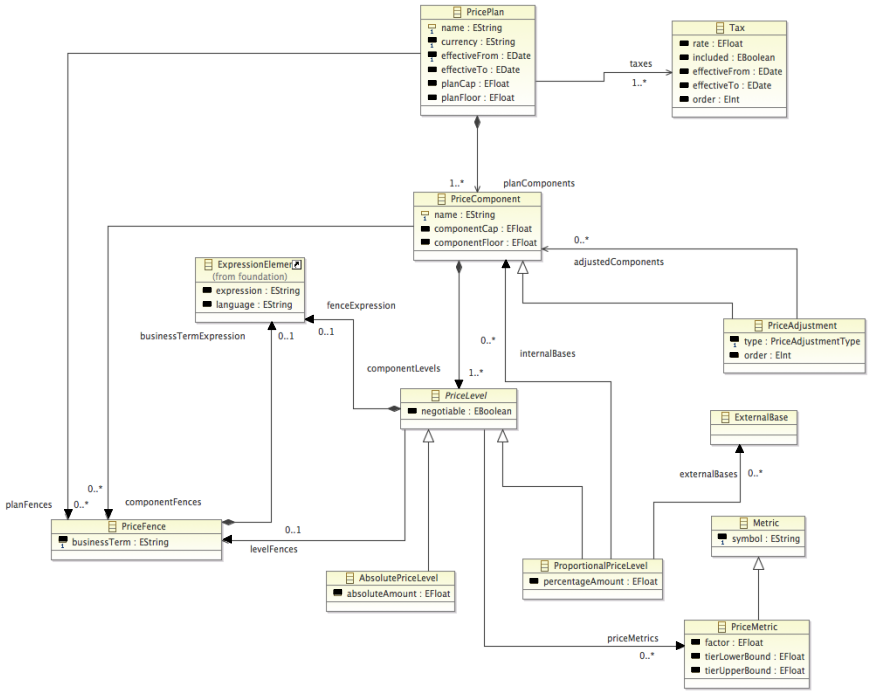


Fig. 4. The Offered variation of the Price Plan metamodel

The requirements for the **Preparation for Offer** state represent a general description of what we call the"relaxed" metamodel. See Figure 3. The idea of relaxing constraints by creating a metamodel variation was introduced by Ramos et. al. [Ramos et al., 2007] to be used in their work on "model snippets" to facilitate pattern matching. Although the delayed validation paradigm of MOF/EMF implies the existence of the relaxed metamodel, the tool chain that we use in RaaS includes the Teneo relational database mapping, which forces a model type check upon save. Therefore we need to explicitly create the relaxed metamodel for use with the repository persistence layer, otherwise partially completed Price models would not be able to be saved into the repository, which is intended as a place for models to be stored for collaborative editing.

The **Offer** state (Figure 4) essentially re-introduces all the lower bound constraints of the original normative metamodel from the USDL standard – however there some small variations, which represent trading conditions for service offered in our example marketplace. However, in this state almost all attributes are unchangeable.

As one might expect, the constraints in the **Deprecated** state are designed to prevent changes, and preserve the Price model as it was when offered to represent a historical record of an advertised service offer (Figure 5).

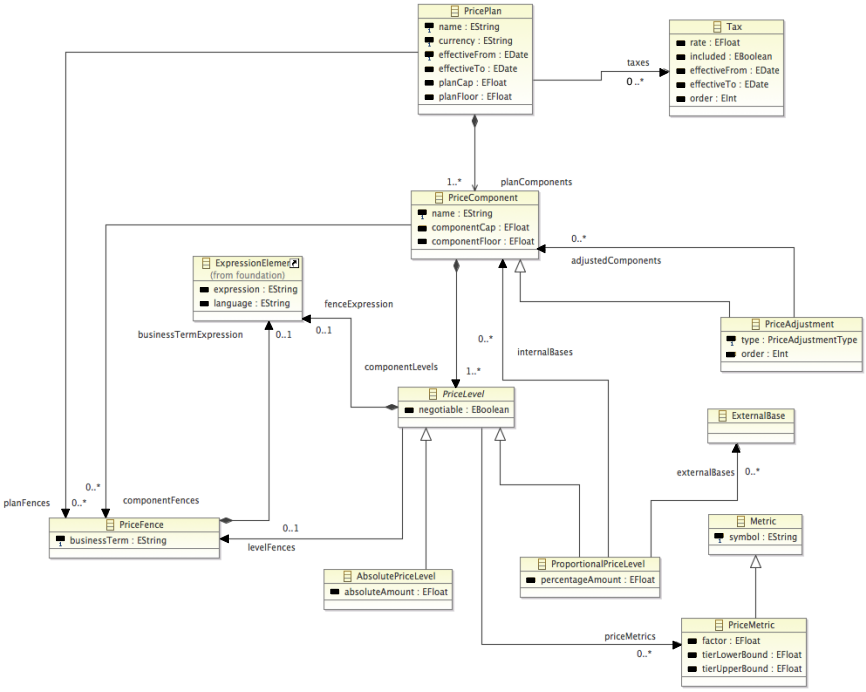


Fig. 5. The Deprecated variation of the Price Plan metamodel

3 Implementation of Variant Constraint Enforcement

The conceptual basis of the approach is that the contents of a repository are modelled as a set of structurally similar metamodels with variations in their meta-attributes which reflect different constraints at different times in a model's lifecycle.

The process by which we achieve a useful set of metamodel variants, and then use them to construct a model repository with state-based constraint enforcement is as follows:

Generate a relaxed version of a standard metamodel. We do this using model transformation in our Tefkat language and engine [Lawley & Steel, 2005]. For our purposes the relaxation of the metamodel is defined as: set all attribute and reference *lowerBound* meta-attributes to zero, and all of their *changeability* meta-attributes to true. We also introduce a simple string-valued attribute in the root class in the relaxed metamodel, *RaaSstate*, to represent the lifecycle state. This attribute will be set by the state machine associated with the instance by the generated code.

Embed a set of annotations into the relaxed metamodel The convention in the EMF tool community for augmenting metamodels for code generation is the use of annotations, which are simply groups of string/string pairs which can be attached to any model element. As they are then embedded into the original metamodel, they can be used via reflection at run time to do constraint checking without additional machinery.

During the construction of the relaxed metamodel, where the new model differs from the normative model, we store the variation as an annotation of the following form:

```
<eStructuralFeatures xsi:type="ecore:EAttribute" xmi:id="2869" name="name"
   eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString">
  <eAnnotations xmi:id="2533" source="lowerBounded">
     <details xmi:id="6135" key="Normative" value="1"/>
  </eAnnotations>
</eStructuralFeatures>
```

The example above shows that the *name* attribute of PricePlan now has the (relaxed) default lowerBound of zero, and that in the normative version the attribute had a lowerBound of one.

Edit the normative metamodel to reflect different constraints. The domain expert that wishes to vary the constraints for each lifecycle state now edits the normative model (or the relaxed model if that is closer to the desired state), and provides a metamodel reflecting the constraints that apply in additional states of a model instance. The tools then apply a diff transformation against the relaxed metamodel which adds additional annotations for embedding that apply to that state. We have a number of diff and patch library transformations written in Tefkat to perform these tasks [Hearnden, 2007]. These are the additional annotations we generated for the *lowerBounded* annotation group for the *name* attribute of PricePlan, as reflected in the metamodels from Figures 3, 4 and 5:

```
<details xmi:id="6136" key="InPreparation" value="0"/>
<details xmi:id="6137" key="Offered" value="1"/>
<details xmi:id="6138" key="Deprecated" value="1"/>
```

Alternatively the domain expert can simply edit the annotations to the relaxed metamodel by copying the *Normative* value *details* already embedded there. The choice here depends on the familiarity of users with particular tools.

Generate code for constraint checking There are many approaches possible, of which we outline the two most appealing here:

1. Generate constraints in an existing constraint language, such as OCL. An OCL constraint can be directly embedded into the Ecore model as annotations to be used by an OCL constraint checking engine. These constraints can be validated both in editors and in the generated Java EMF instances. Here is the example for checking that the name attribute is present (and non-null):

   ```
   invariant lowerBoundedInvariant1:
      self.lifecycleStage = 'Offered' or
      self.lifecycleStage = 'Deprecated'
         implies not self.name.oclIsUndefined() and self.name <> '';
   ```

In later versions of EMF, OCL constraints can also be used to generate Java code rather than executing a constraint checker. This can create very efficient constraint checking implementations. However Eclipes's constraint checking framework does not currently not support triggering validation when setter methods are invoked. Another drawback is that OCL is not capable of accessing the meta-class for an instance so our constraint annotations must be duplicated in the OCL constraints. Therefore we decided to use the following approach.

2. Using the EMF's *generatorAdapters* extension point we embed a generic reflective piece of code which queries the annotations in the metamodel of the object on which the setter is called. The code produces a decision on whether to allow a setter to be executed based on the state of the instance and what is effectively a lookup table in the annotations of its class definition. The presence of our annotations in the metamodel at generation time would cause this code to be embedded. In this way we can extend the approach to enforce additional constraints which vary by lifecycle state during method calls.

The first approach will be more efficient, as the code can execute directly, requiring only the lookup of the current state, and where the cardinalities and changeability are in line with EMF defaults, no guard code will need to be executed. The second is more dynamic, and more maintainable, and allows for information about additional states to be inserted into the annotations at runtime. It will, however, suffer from a slight performance disadvantage due to reflective calls and generic constraint logic.

4 Future Work

Attentive readers will have noted that the state of a service description can transition from **Preparation for Offer** or from **Offered** to the **Deprecated** state. The latter transition seems to contradict the statement that deprecated service offers are preserved for the historical record, as they would become freely editable again if returned to the **Preparation for Offer** state. In fact transitions between states in the fully fledged lifecycle implementation will include the ability to clone models for storage as a record between any two states, and so both transitions from the **Offered** state in the fully functional lifecycle state machine would be marked to store historical copies of the model.

A related matter which is also being addressed is the potential contradiction between required lowerBound multiplicities in the lifecycle state after a transition and the inability in the current state to change the very attributes and references which are required to have additional values. We are investigating several approaches, both of which start by performing an analysis using Tefkat rules on the satisfiability of the constraints in each state from its preceding states. The first approach would report the unsatisfiable constraints to the domain modeller, and require corrections to the metamodel variants. Other approaches include the insertion of an intermediate state with relaxed constraints to allow the model to reach conformance to the next state, or a tool based approach which prompts modellers to provide a minimum set of values to meet the constraints in the new state via an input form or directed editor.

5 Related Work

The techniques for transformation-driven metamodel and model co-evolution of Wachsmuth [Wachsmuth, 2007] are similar to our approach for generating the relaxed metamodel from our normative metamodel, however we preserve all of the class and reference structure of the model, and change only meta-attributes. The implementation for OCL in the EMF framework by Akehust and Patrascoiu [Akehurst & Patrascoiu, 2004] uses the same basic approach as we use for model validation, although their design is for single-state invariant constraint enforcement. We essentially extend this to support the enforcement of invariants that change with the state of the model.

Czarnecki and various co-authors have been modelling variation using feature models for the purposes of creating hardware and software product families. This approach is also extended to deal with cardinalities in [Czarnecki et al., 2005], however, it is assumed that a single instance will be instantiated that falls within the cardinalities specified, and does not vary over time. They unify this approach with metamodelling in [Bak et al., 2010], but the focus is still on product development of multiple instances representing one variation at a time, rather than managing a single data set which has different constraints over time. They document the state of the art in product family variability as at 2010 in [She et al., 2010].

6 Conclusion

We have proposed a mechanism of metamodel variation coupled with a lifecycle state machine which facilitates the generation of constraint checking code in the EMF framework. The approach is designed to allow the improved use of normative specifications which contain metamodels, but which fail to address all of the issues to do with changing constraints over the lifetime of a set of independently evolving model instances. We are augmenting our model repository to tailor the constraints on the models it stores to assist users in keeping their models well-formed during lifecycle changes, by performing extra model validation and modification checking. This approach will be expanded to include consideration of multi-stakeholder modelling processes, such as those found in the service broking and building industries.

References

[Akehurst & Patrascoiu, 2004] Akehurst, David H., & Octavian Patrascoiu 2004. OCL 2.0 - Implementing the Standard for Multiple Metamodels. Electr. Notes Theor. Comput. Sci., 102:21–41.

[Bak et al., 2010] Bak, Kacper, Krzysztof Czarnecki, & Andrzej Wasowski 2010. Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled. In Malloy, Brian A., Steffen Staab, & Mark van den Brand (eds), SLE, volume 6563 of *Lecture Notes in Computer Science*, pages 102–122. Springer.

[Barros & Oberle, 2011] Barros, Alistair, & Daniel Oberle (eds) 2011. Handbook of Service Description – USDL and its Methods. Springer Verlag, Berlin, Germany, 1st edition.

[Cardoso et al., 2009] Cardoso, Jorge, Matthias Winkler, & Konrad Voigt 2009. A Service Description Language for the Internet of Services. In Proceedings First International Symposium on Services Science (ISSS'09), pages 39–48, Berlin, Germany. Logos Verlag.

[Czarnecki et al., 2005] Czarnecki, Krzysztof, Simon Helsen, & Ulrich W. Eisenecker 2005. Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice, 10(1):7–29.

[Duddy et al., 2010] Duddy, Keith, Michael Henderson, Alejandro Metke-Jimenez, & Jim Steel 2010. Design of a model-generated repository as a service for USDL. In Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services, iiWAS '10, pages 707–713, New York, NY, USA. ACM.

[Hearnden, 2007] Hearnden, David 2007. Deltaware: Incremental Change Propagation for Automating Software Evolution in the Model-Driven Architecture. PhD thesis, University of Queensland.

[Kiemes & Oberle, 2010] Kiemes, Tom, & Daniel Oberle 2010. Generic Modeling and Management of Price Plans in the Internet of Services. In Fähnrich, Klaus-Peter, & Bogdan Franczyk (eds), GI Jahrestagung (1), volume 175 of *LNI*, pages 533–538. GI.

[Lawley & Steel, 2005] Lawley, Michael, & Jim Steel 2005. Practical Declarative Model Transformation with Tefkat. In Bruel, Jean-Michel (ed), Satellite Events at the MoDELS 2005 Conference, Revised Selected Papers, volume 3844 of *LNCS*, pages 139–150, Berlin, Germany. Springer Verlag.

[Object Management Group, 2008] Object Management Group 2008. Meta Object Facility (MOF) Core Specification Version 2.4. OMG Document No. formal/2008-12-10.

[Ramos et al., 2007] Ramos, Rodrigo, Olivier Barais, & Jean-Marc Jézéquel 2007. Matching Model-Snippets. In Engels, Gregor, Bill Opdyke, Douglas Schmidt, & Frank Weil (eds), Model Driven Engineering Languages and Systems, volume 4735 of *Lecture Notes in Computer Science*, pages 121–135. Springer Verlag, Berlin, Germany.

[She et al., 2010] She, Steven, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, & Krzysztof Czarnecki 2010. The Variability Model of The Linux Kernel. In Benavides, David, Don S. Batory, & Paul Grünbacher (eds), VaMoS, volume 37 of *ICB-Research Report*, pages 45–51. Universität Duisburg-Essen.

[Steel et al., 2010] Steel, Jim, Robin Drogemuller, & Bianca Toth 2010. Model interoperability in building information modelling. Software and Systems Modeling, pages 1–11–11.

[Succar, 2009] Succar, B. 2009. Building information modelling framework: A research and delivery foundation for industry stakeholders. Automation in Construction, 18(3):357–375.

[Wachsmuth, 2007] Wachsmuth, Guido 2007. Metamodel Adaptation and Model Co-adaptation. In Ernst, Erik (ed), ECOOP, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624. Springer.

Towards Semantics-Aware Merge Support in Optimistic Model Versioning*

Petra Brosch[2], Uwe Egly[1], Sebastian Gabmeyer[2], Gerti Kappel[2], Martina Seidl[3], Hans Tompits[1], Magdalena Widl[1], and Manuel Wimmer[2]

[1] Institute for Information Systems, Vienna University of Technology, Austria
{uwe,tompits,widl}@kr.tuwien.ac.at
[2] Business Informatics Group, Vienna University of Technology, Austria
{lastname}@big.tuwien.ac.at
[3] Institute of Formal Models and Verification, Johannes Kepler University, Austria
martina.seidl@jku.at

Abstract. Current optimistic model versioning systems, which are indispensable to coordinate the collaboration within teams, are able to detect several kinds of conflicts between two concurrently modified versions of one model. These systems support the detection of syntactical problems such as contradicting changes, violations of the underlying metamodel, and violations of OCL constraints. However, violations of the semantics remain unreported. In this paper, we suggest to use redundant information inherent in models to check if the semantics is violated during the merge process. In particular, we exploit the information encoded in state machine diagrams to validate evolving sequence diagrams by means of the model checker SPIN.

1 Introduction

In model-driven engineering, *version control systems* (VCS) are an essential tool to manage the evolution of software models [4]. In this respect, *optimistic version control systems* [1] are of particular importance. They provide reliable recovery mechanisms in case changes have to be undone and support the collaboration of multiple developers.

An optimistic VCS stores the artifacts under development in a central repository, which may be accessed by all team members at any time. A typical interaction with the repository starts when a developer checks out the most recent version of the model under development. The developer then performs the desired changes on a local copy. Upon completion, the developer checks the modified local version back into the repository. If the performed changes do not interfere with the concurrently introduced modifications of another developer, the merge is straightforward and may be computed automatically. Otherwise, a *merge conflict* [4] is at hand and the divergent versions need to be merged

* This work was partially funded by the Austrian Federal Ministry of Transport, Innovation, and Technology and the Austrian Research Promotion Agency under grant FIT-IT-819584, by the Vienna Science and Technology Fund (WWTF) under grant ICT10-018, by the fFORTE WIT Program of the Vienna University of Technology and the Austrian Federal Ministry of Science and Research, and by the Austrian Science Fund (FWF) under grants P21698 and S11409-N23.

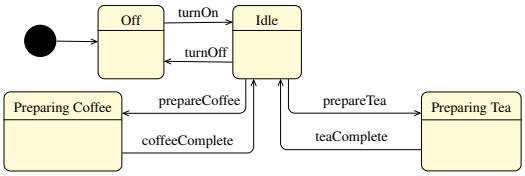


Fig. 1: State machine diagram for the class CoffeeMachine (`CM`).

manually. Without adequate tool support, the merged version may result in a syntactically and/or semantically inconsistent version, even though both versions were consistent prior to the merge. Obviously, it is of paramount importance to detect and resolve conflicts as soon as possible to prevent their propagation through multiple development cycles.

Among the many possible merge conflicts [1], the most common are *contradicting changes*. Given two developers working on the same model, this conflict may emerge if both developers commit their changes and either (a) their changes may not be applied in combination (i.e., *delete/update*), or (b) their changes are not commutable (i.e., *update/update*). In the latter case, the different ordering of the changes results in different models. In such a situation, often user interaction is required to resolve the conflict. Alternatively, a predefined heuristic-based merge strategy may be applied to automatically generate consolidated, syntactically correct versions. However, it cannot be asserted that the model is *semantically* consistent.

Consider the following example, which describes a semantically inconsistent model caused by an automatic merge of changes. Figure 1 depicts a UML state machine diagram modeling a coffee machine and the upmost diagram S in Fig. 2 a possible behavior of the same machine in terms of a sequence diagram. Two software engineers change the sequence diagram at the same time: one includes the message $turnOff()$, resulting in S', the other adds the message $prepareTea()$, resulting in S''. Each change on its own results in a sequence diagram consistent with the state machine. The next step is to merge the changes into a new sequence diagram $\hat{S}$ using an automatic versioning tool, e.g., as the one proposed by Brosch et al. [2]. As the messages of a lifeline are represented as ordered list, an update/update conflict occurs, because both newly added messages are stored at the same index of this list. A conceivable merging strategy is to consider all possible combinations of the two diagrams. This may result in several syntactically correct diagrams. Figure 2 shows two possibilities, $\hat{S}_1$ and $\hat{S}_2$: $turnOff()$ can be placed before or after $prepareTea()$. However, making tea after turning off the machine does not make much sense and a modeler would avoid such a solution in a manual merge process.

At first glance, it might seem necessary to provide additional knowledge, e.g., a specifically tailored ontology, to support an automatic merge process aware of the model's semantics. However, in modeling languages like UML, the required knowledge is distributed over different types of diagrams. Each diagram type provides a view on a specific aspect of the described system. Yet, these views overlap in parts, effectively duplicating certain aspects of the system across different diagrams. For our example, we may ascertain, that the first merge option, i.e., $turnOff()$ before $prepareTea()$, turns

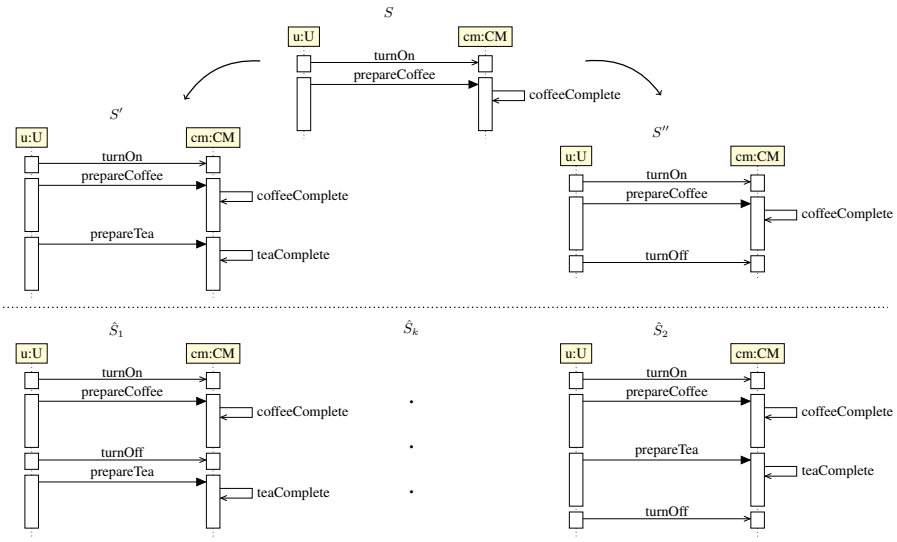


Fig. 2: Versioning scenario for a sequence diagram.

out to be inconsistent with respect to the state machine diagram, as preparing tea after turning off the machine is not possible.

In this paper, we thus propose to exploit this redundant information and use the overlapping parts of the diagrams as gluing points to construct a coherent picture of the system. In this way, we are able to assert that the modifications performed on a sequence diagram are consistent with the specification stated in a state machine diagram. For this purpose, we employ model checking techniques and integrate this approach into the merging component of the model versioning system AMOR [2].

Starting with a review of related work in Section 2, we proceed to present our semantics-aware merging approach in Section 3. We then showcase how the above presented example is solved with our approach in Section 4. Section 5 concludes the paper with a discussion of future work directions.

2 Related Work

The fields of model versioning and model verification are both related to our work, which we describe in what follows.

Model Versioning. In the last decade more than a dozen model versioning systems have been proposed (see [1] for an overview). Many existing model versioning systems take advantage of the graph-based structure of software models. As a consequence, conflicts resulting from contradicting changes are more precisely detected, sometimes even automatically resolved. Since changes are rarely introduced independently of each other, think of refactorings for example, some approaches analyze the set of composite changes

to recognize the user's intention, and try to derive suitable resolution strategies when conflicting versions are checked into the repository [2, 6]. However, the semantic aspects of models are mostly neglected by current model versioning systems. To the best of our knowledge, only two approaches consider semantics in the context of model versioning. The first approach suggests the usage of *semantic views*, which are constructed by a manually defined normalization process that removes all duplicate representations of one and the same concept from the original metamodel [13]. When two divergent versions of the same base model are checked into the repository, the two versions are normalized and compared to determine possible conflicts. Although the normalization procedure integrates a semantic layer into the model versioning process, the actual comparison of the normalized models is still performed on a syntactic level.

Another elegant technique, which employs *diff* operators to compare models, is presented by Maoz et al. [10]. A *diff* operator $diff(m_1, m_2)$ expects two models, m_1 and m_2, as input and outputs a set of so-called diff witnesses, i.e., instances of m_1 which are not instances of m_2. For example, two *syntactically* different models m_1 and m_2 are *semantically* equivalent if each instance of m_1 is an instance of m_2 and vice versa. While [10] focuses solely on the semantic differencing aspect of model versioning, we aim to advance to a semantics-aware model merging process that is supported by an inter-diagram based consistency verification technique.

Model verification. Decoupled from the above sketched research field of model versioning systems, various works propose the verification of the syntactical consistency of models, many of which focus on the verification of UML diagrams (e.g. [7, 11]). The verification process may be enhanced by the addition of semantic information. For example, Cabot et al. [5] verify the behavioral aspects of UML class diagrams annotated with so-called operation contracts, which are declarative descriptions of operations specified as OCL pre- and postconditions. The class diagram and the operation contracts are thereby transformed into a constraint satisfaction problem, which is solved with respect to a set of consistency properties expressing, e.g., the applicability or the executability of an operation. A formal verification technique for UML 2.0 sequence diagrams employing linear temporal logic (LTL) formulas and the SPIN model checker [8] to reason about the occurrences of events is introduced by Lima et al. [9]. In contrast to these single-diagram verification techniques, multi-view approaches assert the consistency across a set of diagrams. Proponents in this area are, among others, the tools HUGO [14] and CHARMY [12]. HUGO verifies whether the interactions of a UML collaboration diagram are in accordance with the corresponding set of state machine diagrams. The tool automatically translates the state machine diagrams to PROMELA, the input language of SPIN, and generates so-called "never claims" from the collaboration diagrams. The generated artifacts form the input for SPIN, which performs the verification. While HUGO operates on UML diagrams, CHARMY provides a modeling, simulation, and verification environment for software architectures (SA), which share many commonalities with UML. SAs describe the static and behavioral structures of systems with component, state transition, and sequence diagrams. Again, CHARMY translates the modeled artifacts to PROMELA and calls upon SPIN to either locate deadlocks and unreachable states in the state machines, or to verify temporal properties of the system. In contrast to the standalone, snapshot-based verification procedure implemented by CHARMY and HUGO,

our approach integrates the consistency verification procedure into the model versioning process to enable the semantics-aware merging of models.

3 Semantics-Aware Model Versioning

To detect semantic merge problems as described above, we propose to use a model checker like SPIN [8] within the merge process. The idea is to generate possible merge results and to check for each if it is consistent with the behavior defined by the corresponding state machine. We first give a short definition of the modeling language concepts needed, and then introduce our approach in detail. In particular, we consider a simplified subset of the UML state machine and sequence diagrams.

3.1 Definitions

For our purposes, a *software model* $\mathcal{U}$ consists of a set $\mathcal{M}$ of *state machines* and a *sequence diagram* $\mathcal{S}$, defined as follows: A state machine $M = (Q, T, \tau, v_0, A)$ is a deterministic finite automaton, where

- Q is a set of *states*,
- T is a set of *transition labels* (or possible *input symbols*),
- $\tau : Q \times T \to Q$ is the *transition function*,
- $q_0 \in Q$ is a designated *initial state*, and
- $A \subseteq Q$ is a set of *accepting states*.

A sequence diagram $\mathcal{S}$ is a tuple $(N, \overline{\mathcal{L}})$, where N is a set of *messages* and $\overline{\mathcal{L}}$ is a set $\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$ of *lifelines*. A lifeline, $\mathcal{L}$, in turn, is a tuple (M, L, tr), where

- $M \in \mathcal{M}$ is a state machine,
- L is a finite sequence $(n_1, \ldots, n_m)$ of elements of N and
- $tr : N \to T$ is a bijective function, mapping each message to a transition of the corresponding state machine.

A model $\mathcal{U}$ is *consistent* iff for each lifeline $\mathcal{L} = (M, L, tr)$ of $\mathcal{S}$, there exists a path $(tr(n_1), \ldots, tr(n_m))$ in the state machine M, where $L = (n_1, \ldots, n_m)$.

3.2 Versioning Scenarios

Our versioning scenarios involve concurrent modifications on a sequence diagram. The state machine diagrams remain unchanged. A modification concerns one or more messages, each being of either of the following three types:

- *insert*: a message $n \in N$ is inserted at any index of a lifeline;
- *delete*: a message n is removed from a lifeline; and
- *update*: a message n is replaced by $n' \in N$ different from n.

Concurrent changes may result in different sequence diagrams. It is then up to the versioning tool to merge these changes into a new version of the diagram, which must be syntactically correct and consistent with the state machine diagrams.

Merging sequence diagrams is done as follows: For each lifeline, any possible sequence of messages originating from both diagrams is syntactically correct, but possibly inconsistent with the behavior defined in the corresponding state machine diagrams.

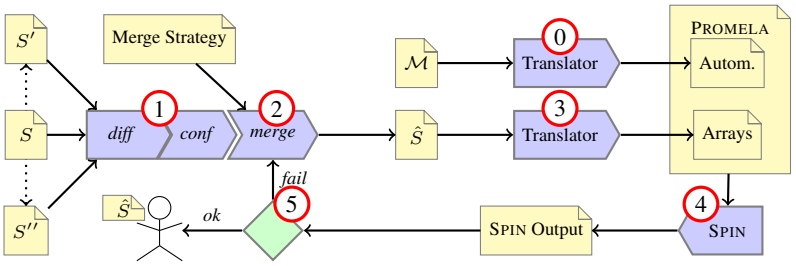


Fig. 3: Workflow of the merging process.

3.3 Semantics-Aware Model Merging

We propose to integrate the model checker SPIN [8] to support the generation of merged sequences. SPIN is a software verification tool: It takes as input a software abstraction, or model, encoded in SPIN's input language PROMELA and relevant properties of the software model in LTL. SPIN can be run in two modes: In *simulation mode*, where the PROMELA model is executed, and in *verification mode*, where the LTL formula is checked for satisfiability with respect to the PROMELA model.

For our basic definition of a software model, we propose a simple encoding that allows to check for the consistency between a sequence diagram and a set of state machines by running SPIN in simulation mode, which is much faster than verification mode and sufficient for our purpose. The state machines are encoded as deterministic finite automata and the sequence diagram as set of arrays containing transition labels of the respective automata. The verification task in this case is to check if each word (i.e., array of transition labels) is accepted by its automaton.

The workflow of the merging process, as depicted in Fig. 3, is as follows:

0. The set $\mathcal{M}$ of state machine diagrams is encoded in PROMELA automata. Other than for sequence diagrams, this encoding is done only once per application scenario.
1. The versioning operations *diff* (comparison) and *conf* (conflict detection) are executed on the original sequence diagram S and the two modifications S' and S''.
2. The versioning operation *merge* is performed based on the output of Step 1 and a merge strategy. In order to produce a syntactically correct sequence diagram, the merge strategy defines conditions on the possible orderings of the merged messages on a lifeline. A possible strategy is one that orders messages in a first-come, first-serve manner, or one that allows any possible combination. A strategy may allow more than one possible sequence diagram as result of the merge. In this case, the choice is made deterministically.
3. The output of *merge*, i.e., a syntactically correct sequence diagram $\hat{S}$, is encoded as set of PROMELA arrays, describing each lifeline as a word from the alphabet of the respective automaton encoded in Step 0.
4. The PROMELA code is fed into SPIN, which checks if each of the words generated in Step 3 is accepted by the respective automaton. It returns either a success message or the state and transition where the verification failed.

5. If the SPIN output does not contain an error message, the current merged sequence diagram $\hat{S}$ and the SPIN output are returned to the user. Otherwise, the procedure continues at Step 2 with a new merged sequence diagram $\hat{S}$ different from the previous ones.

For the encoding we make use of the following elements of PROMELA [8]:

- `active proctype`: defines a process behavior that is automatically instantiated at program start;
- `label`: identifies a unique control state (we also use the prefix `end`, which defines a termination state);
- `mtype`: a declaration of symbolic names for constant values;
- `array`: a one-dimensional array of variables (we use arrays of `mtype` elements to encode words checked by the automaton);
- `if`: a selection construct, used to define the structure of the automaton; and
- `goto`: an unconditional jump to a label, also used to define the structure of the automaton.

The PROMELA encoding of a state machine is done as follows:

- The state machine is encoded as `active proctype` that contains all the necessary elements of the state machine.
- Each transition $t \in T$ is encoded as an element of `mtype`. The additional element `acc` is added to model transitions to the `end` state.
- Each $q \in Q$ is encoded as a label marking a state of the `active proctype`. The additional state `end` is added.
- The state q_0 is placed at the beginning of the respective process in order to be executed at process initiation.
- τ is encoded as a set of `if` conditions inside each PROMELA state q: For each t such that (q, t) is defined by τ, the current symbol of an input sequence (which is, as described below, the encoding of a lifeline) is compared to t. If the condition holds, a `goto` statement jumps to state $\tau(q_0, t)$.
- Our sequence diagram semantics does not require a lifeline to end with a specific message, so all states are accepting states. We thus place a transition `goto end` if the current symbol equals our additional transition label `acc` into each state except the `end` state.

A lifeline is encoded as array `S` of `mtype`. Each field of `S` with index `i` contains the `mtype` element $tr(e_i)$ where e_i is the i-th element of the sequence L.

The PROMELA code is executed as simulation. It prints a success message if the word encoded in the array is accepted. In this case, the lifeline is consistent with the corresponding state machine. Otherwise it aborts when it hits a transition label that is undefined in the current state.

We have implemented the outlined approach based on the Eclipse Modeling Framework (EMF)[4]. In particular, the presented language excerpt of UML has been specified as an Ecore-based metamodel. The transformations of state machines into PROMELA

[4] `http://www.eclipse.org/modeling/emf`.

automata and sequence diagrams into PROMELA arrays have been implemented as model-to-text transformations using Xpand[5]. The implementation is available at `http://modelevolution.org`.

4 Application Scenario

We illustrate our approach using the example from Section 1. First, we translate the state machine of Fig. 1 by means of the encoding presented in the previous section as follows:

- The state machine is defined as `active proctype` named `Coffeemachine`.
- The transition labels of the coffee machine, along with an additional label `acc`, are contained in `mtype`.
- Each state of the coffee machine is represented by a label, such as `Off` or `Idle`, and an `end` state is added. The start and end states of the coffee machine are summarized in label `Off`.
- For each state, all defined transitions are encoded using `if` and `goto` statements.
- A counter is added to keep track of the current index of the input word.

Listing 4.1: State machine encoding in PROMELA.

```
1  mtype = {turnOff,turnOn,prepareCoffee,coffeeComplete,prepareTea,teaComplete,
2           acc};
3
4  active proctype Coffeemachine() {
5  byte h = 0;
6  mtype CM[3];
7
8  CM[0] = turnOn; CM[1] = prepareCoffee; CM[2] = coffeeComplete; CM[3] = acc;
9
10 Off:
11  printf("Off\t %e\n", CM[h]);
12  if
13  :: CM[h] == turnOn -> h++; goto Idle
14  :: CM[h] == acc -> goto end
15  fi;
16 Idle:
17  printf("Idle\t %e\n", CM[h]);
18  if
19  :: CM[h] == prepareCoffee -> h++; goto PreparingCoffee
20  :: CM[h] == prepareTea -> h++; goto PreparingTea
21  :: CM[h] == turnOff -> h++; goto Off
22  :: CM[h] == acc ->  goto end
23  fi;
24 PreparingCoffee:
25  printf("PreparingCoffee\t %e\n", CM[h]);
26  if
27  :: CM[h] == coffeeComplete -> h++; goto Idle
28  :: CM[h] == acc -> goto end
29  fi;
30 PreparingTea:
31  printf("PreparingTea\t %e\n", CM[h]);
32  if
33  :: CM[h] == teaComplete -> h++;  goto Idle
34  :: CM[h] == acc -> goto end
35  fi;
36 end:
37  printf("end!\n")
38 }
```

[5] `http://www.eclipse.org/modeling/m2t/?project=xpand`.

The sequence diagram contains one relevant lifeline, the instance cm of the coffee machine, which is encoded as array `CM` of `mtype`: For each message n_i received by cm, `CM[i]=` $tr(n_i)$. Recall that tr returns an element of the set of transition labels and that those are encoded as elements of `mtype`.

The resulting encoding of the state machine with the initial version S of the sequence diagram is shown in Listing 4.1. It is easy to see that the above code eventually reaches the `end` state. Replacing the array `CM` by the two modified sequence diagrams S' and S'', encoded in the same manner, the code also reaches the `end` state. However, on the merged sequence diagram $\hat{S}_1$, given in the following, the model checker will give up when it reaches the `Off` state trying to match `CM[4]`.

```
6  mtype CM[7];
7  CM[0] = turnOn; CM[1] = prepareCoffee; CM[2] = coffeeComplete; CM[3] = turnOff;
8  CM[4] = prepareTea; CM[5] = teaComplete; CM[6] = acc;
```

On the other hand, the second merged sequence diagram $\hat{S}_2$, given in the following, is consistent. Hence, in our merging workflow, $\hat{S}_2$ will be returned to the user.

```
6  mtype CM[7];
7  CM[0] = turnOn; CM[1] = prepareCoffee; CM[2] = coffeeComplete;
8  CM[3] = prepareTea; CM[4] = teaComplete; CM[5] = turnOff; CM[6] = acc;
```

5 Conclusion and Future Work

In this paper, we proposed to use a model checker to detect semantic merge conflicts in the context of model versioning. Model checkers are powerful tools used for the verification of hardware and software. A model checker takes as input a model of a system and a formal specification of the system and verifies if the former meets the latter. We applied this technique to check the semantic consistency of an evolving UML sequence diagram with respect to state machine diagrams that remain unchanged. When contradicting changes occur, a unique automatic merge is not possible in general. However, additional information on violations of the model's semantics allows to identify invalid solutions. Hence, a more goal-oriented search for a consistent merged version is supported.

Our first experiments on this approach gave promising results, but for the full integration into the versioning process several issues have to be considered which we discuss in the following.

Extension of the Language Features. So far, we considered only a restricted, simplified subset of the UML metamodel. In this setting, the execution semantics of the considered diagrams is quite unambiguous. With the introduction of more advanced concepts, several questions concerning the execution semantics will arise, which are not covered by the UML standard and need detailed elaboration in order to translate them to the formalisms supported by the model checker. When including these language features, we expect to fully exploit the expressiveness of LTL for the needed assertions.

Integration in the Merge Component. We use the information obtained by the model checker not only to verify the consistency of two diagrams, but to support the merge process as necessary when models are versioned in an optimistic way. At the moment, only the fact that the model checker failed to verify the provided encoding is propagated

back to the merge component. We plan to build an analyzer which is able to deduce constraints from the output of the model checker. These constraints can then be used to create an alternative merged version.

Visualization of the Conflicts. For reasons of usability, the representation of conflicts is of paramount importance. In particular, we conjecture that conflicts have to be reported in the concrete syntax of the modeling language [3]. Therefore, we propose a mechanism based on UML profiles to include merging information directly into the model. We plan to extend this mechanism to report semantical problems in the concrete syntax.

Benchmarking. Finally, we need more test cases to evaluate our approach. In particular, it will be interesting to learn about precision and recall in various merging scenarios as well as to study scalability with growing model size.

References

1. P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. The Past, Present, and Future of Model Versioning. In *Emerging Technologies for the Evolution and Maintenance of Software Models*. IGI Global, 2011.
2. P. Brosch, G. Kappel, M. Seidl, K. Wieland, M. Wimmer, H. Kargl, and P. Langer. Adaptable Model Versioning in Action. In *Modellierung*, volume 161 of *LNI*, pages 221–236. GI, 2010.
3. P. Brosch, H. Kargl, P. Langer, M. Seidl, K. Wieland, M. Wimmer, and G. Kappel. Conflicts as First-Class Entities: A UML Profile for Model Versioning. In *Models in Software Engineering*, volume 6627 of *LNCS*, pages 184–193. Springer, 2011.
4. P. Brosch, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. Colex: A Web-based Collaborative Conflict Lexicon. In *IWMCP @ TOOLS'10*, pages 42–49, 2010.
5. J. Cabot, R. Clarisó, and D. Riera. Verifying UML/OCL Operation Contracts. In *7th Int. Conf. on Integrated Formal Methods*, pages 40–55. Springer, 2009.
6. A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing Model Conflicts in Distributed Development. In *11th Int. Conf. on Model Driven Engineering Languages and Systems, MoDELS'08*, volume 5301 of *LNCS*, pages 311–325, 2008.
7. A. Egyed. UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models. In *29th Int. Conf. on Software Engineering*, pages 793–796. IEEE, 2007.
8. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
9. V. Lima, C. Talhi, D. Mouheb, M. Debbabi, L. Wang, and M. Pourzandi. Formal Verification and Validation of UML 2.0 Sequence Diagrams using Source and Destination of Messages. *ENTCS*, 254:143–160, 2009.
10. S. Maoz, J. O. Ringert, and B. Rumpe. A Manifesto for Semantic Model Differencing. In *Models in Software Engineering*, volume 6627 of *LNCS*, pages 194–203. Springer, 2010.
11. T. Mens, R. Van Der Straeten, and M. D'Hondt. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In *9th Int. Conf. on Model Driven Engineering Languages and Systems, MoDELS'06*, volume 4199 of *LNCS*, pages 200–214. Springer, 2006.
12. P. Pelliccione, P. Inverardi, and H. Muccini. CHARMY: A Framework for Designing and Verifying Architectural Specifications. *TSE*, 35(3):325–346, 2008.
13. T. Reiter, K. Altmanninger, A. Bergmayr, W. Schwinger, and G. Kotsis. Models in Conflict – Detection of Semantic Conflicts in Model-based Development. In *MDEIS @ ICEIS'07*, pages 29–40, 2007.
14. T. Schäfer, A. Knapp, and S. Merz. Model Checking UML State Machines and Collaborations. *ENTCS*, 55(3):357–369, 2001.

Domain Specific Language Modeling Facilities

Jean-Philippe Babau and Mickaël Kerboeuf

LISyC, UBO, UEB
{babau,kerboeuf}@univ-brest.fr

Abstract. This paper describes a metamodel called Modif dedicated to metamodel evolution description. Evolutions are common editing (remove, rename and change), refactoring (flatten) and an original *hide* operator to apply to Ecore elements. From a Modif model, tools automatically generate the target metamodel and corresponding model migrations. The approach is illustrated on a specific Finite State Machine metamodel defined as an evolution of existing UML concepts.

1 Introduction

Model Driven Engineering (MDE) proposes an intensive usage of models to implement Software Engineering principles. To implement a MDE process, designers have to master model definition and model handling.

For modeling activities, a General Purpose Modeling Language, such as UML [12], provides rich, but large, complex, and non specific, existing modeling concepts. Whereas, a Domain Specific Modeling Language is handier for specific concerns. In this paper, we focus on Ecore DSML definition using Eclipse Modeling Framework (EMF) [3].

With regard to the design of DSML, in the same domain, most of them share many common concepts. They vary only by a limited set of concepts, naming and structural considerations. For example, we may easily find dozens of Ecore models of specific Finite State Machine sharing the same concepts more or less. Thus a new DSML is often an evolution of an existing modeling language.

Introducing a new DSML leads to producing yet another modeling language. It induces a *Babel Tower problem*, *i.e.* a lot of models that are difficult to manage, and the need to build interoperability bridges between them through model transformations, called model migrations [13]. To tackle this problem, we need tools to automate DSML definition and model migrations.

The aim of this paper is to facilitate DSML definition and manipulation by proposing a metamodel, called *Modif*, to explicit metamodel evolution. Then, tools based on Modif encompass metamodel generation from legacy metamodels, and automatic generation of model migrations between source and target models.

This paper is organized as follows. The next section presents Modif metamodel and proposed transformations (at meta and model levels). The following section shows an experiment where we quickly implement a tool to produce specific FSM models as an evolution of UML models. We finally conclude the paper and give some perspectives.

2 DSML tooling

2.1 Principles

In our approach, we consider a legacy metamodel (MMs.ecore in figure 1). From it, the DSML designer has only to produce a model (MMs2MMt.modif) which gathers modifications that should be applied on MMs to produce the target metamodel (MMt.ecore). Then, the kermeta [11] transformation modifMM.kmt automatically produces MMt.ecore from both MMs.ecore and MMs2MMt.modif.

The main idea of Modif is to associate to each Ecore concept of MMs some classical editing operators (*remove*, *rename* and change attributes) to get MMt. Moreover, some classical refactoring operators are provided (*flatten*). And, because *remove* is sometimes too strong, a specific, and original, *hide* operator is proposed for EClasses and EPackages.

Based on MMs2MMt and MMs, another kermeta transformation generateM2M.kmt generates a model migration Ms2Mt.kmt. It enables transformation of MMs model to MMt model.

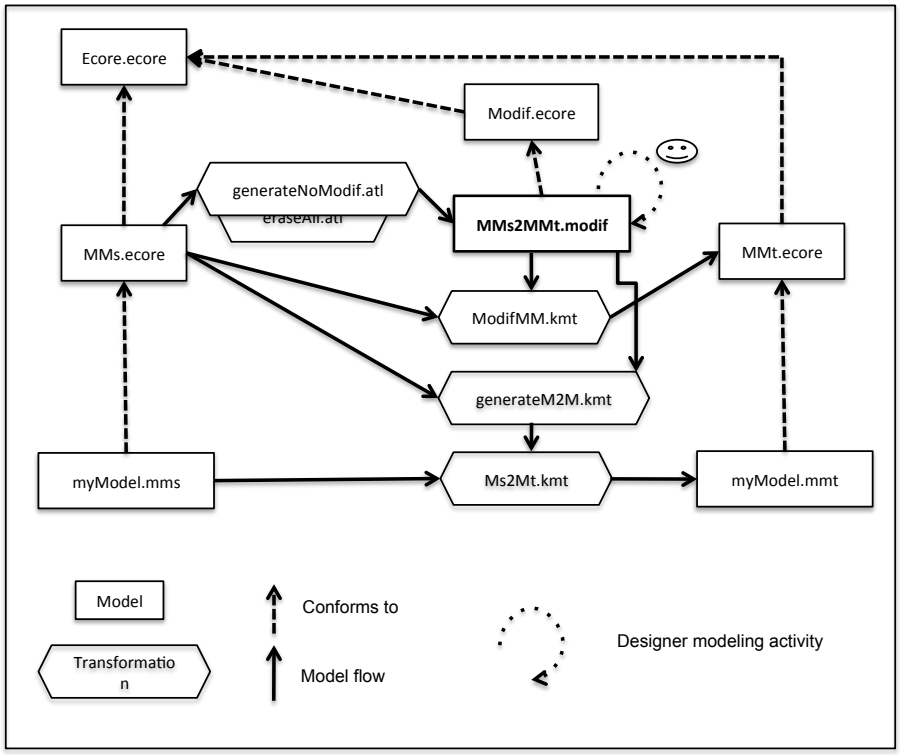


Fig. 1. Modif tooling process

Because of the way Modif is processed (detailed below), it may be necessary to apply the operations many times to obtain the expected target MMt.

Below we now give details of Modif metamodel (see figure 2) and associated transformations.

2.2 Modif

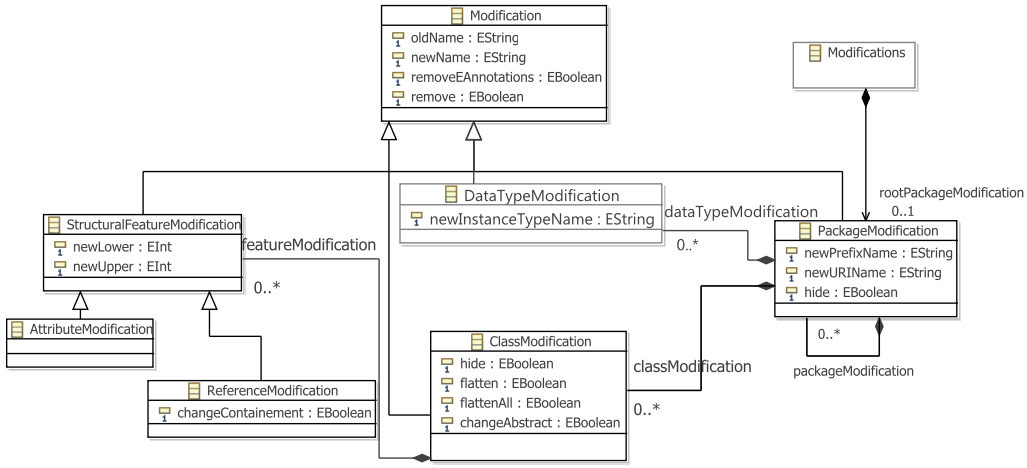


Fig. 2. graphical view of metamodel Modif.ecore

First, because the goal of Modif is to build a MMt from an existing MMs, we consider that all the necessary concepts are defined in MMs, so we do not provide an *add* operator. We can separate Modif operators into 3 families:

1. *Deleting operators*: the source concept is not present in the target
2. *Refactoring operators*: the concept is structured differently in the target
3. *Changing operators*: the Ecore property of the source concept is modified

Deleting operators are specified when the flags *remove* or *hide* are set to value *true*. *Remove* takes precedence over *hide*: if *remove* is set to *true*, then *hide* is not considered. In the same way, deleting operators take precedence over modifying and refactoring operators: if *remove* or *hide* are set to *true*, modifications and refactoring are not considered.

There follows an explanation of how this data is considered at a metamodel level, and so how the target MMt is built from MMs and MMs2MMt.modif.

Deleting operators Case *remove* is *true*: the source concept disappears and we apply the following rules:

- for all kinds of concepts: delete contained EAnnotations;
- for an EDataType ED: also remove all EAttributes where EType is ED;
- for an EClass EC: also remove all EStructuralFeatures contained in EC and all the EReferences ER to EC (ER.*EType* = EC);
- for an EPackage EP: also remove all the EObjects contained in EP.

The *hide* operator is trickier. Before giving details of their impact, we introduce the notion of *inheritance path* and *reference path*. These are based on the EClass graph defined by an Ecore metamodel where EClasses are considered as nodes, and associations (EReference and Inheritance) are considered as edges.

An *inheritance path* exists between two EClasses EC1 and EC2, if they are connected by a set of inheritance edges (EC1 explicitly or implicitly inherits from EC2). An *inheritance path* is made up of a set of intermediate connected EClasses. In figure 3, (C,B) is an *inheritance path* between D and A.

In the same way, a *reference path* exists between EC1 and EC2, if EC1 and EC2 are connected by a set of EReference edges (from an instance of EC1, it is possible to directly or indirectly refer to an instance of EC2). A *reference path* is made up of a set of corresponding EReference and intermediate connected EClasses. In figure 3, (ref_B, B, ref_F) is a *reference path* between E and F.

We now give details of the effect of hide on EPackages and EClasses :

- for an EPackage EP: EP is deleted and all contained EObjects that are not removed and not hidden are moved to EP.*container*;
- for an EClass EC:
  - EC is deleted and all EStructuralFeatures contained in EC and all the EReferences ER to EC (ER.*EType* = EC) are also removed;
  - an inheritance relationship between EC1 and EC2 is added if a non empty *inheritance path* made up of only hidden EClasses exists between EC1 and EC2;
  - an EReference from EC1 to EC2 is added if a non empty *reference path* P made up of only hidden EClasses exists from EC1 to EC2 with the following properties:
    - its *name* is a concatenation of the names of all paths' elements
    - the *containment* property is *true*, if and only if, all the containment properties of each EReference of P is *true*
    - the *lowerBound* (resp. *upperBound*) value is equal to the product of each *lowerBound* (resp. *upperBound*) value of each EReference of P.

A *hide* operator is a weak implementation of a *remove* operator keeping implicit links between EClasses. To illustrate this point, we apply a *hide* operator to EClass B of MMs (see figure 3). This produces the metamodel MMt1 presented on figure 4. We see that if we remove B, the inheritance relationship between C and A is preserved, and a reference is added between E and F. In the two metamodels, an instance of E may be directly or indirectly connected to 0 or 1 instance of F. We have only illustrated some property preservations. Transformation semantic is not the subject of this paper and not detailed here.

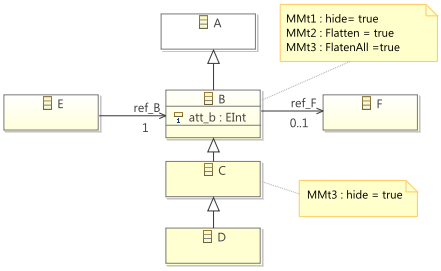


Fig. 3. graphical view of metamodel source MMs

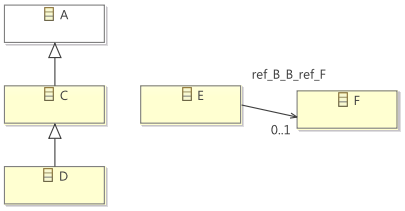


Fig. 4. graphical view of metamodel target MMt1

Refactoring operators Case *flatten* is *true* for a given EClass EC:

- all the not deleted EStructuralFeature of EC are duplicated on all EClasses subEC that EC is a supertype of subEC;
- the inheritance relationship between subEC and EC is removed;
- each EReference ER from an EClass othEC to EC is duplicated to a EReference from othEC to subEC, and is renamed (ER.*name* + '_' + subEC.*name*);

Case *flattenAll* is *true* for a given EClass EC : the same operations as for flatten are performed but they are applied to all non hidden subEC where a non *inheritance path*, even empty, made up of only hidden EClasses between subEC and EC exists. The attribute *flattenAll* takes precedence over the attribute *flatten*: if *flattenAll* is *true*, then *flatten* is not considered.

To illustrate *flatten* operators, we first set *flatten* to *true* for B. It produces the metamodel MMt2 (see figure 5). Then we set *flattenAll* to *true* for B and we set *hide* to *true* for C. It produces the metamodel MMt3 (see figure 6). We note that an EReference is added from E to keep a relationship with C, resp D. We also note that it is possible to combine *hide* and *flatten* (through *flattenAll*), thus reducing the number of transformation operations.

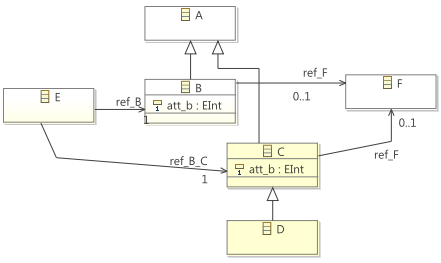


Fig. 5. graphical view of metamodel target MMt2

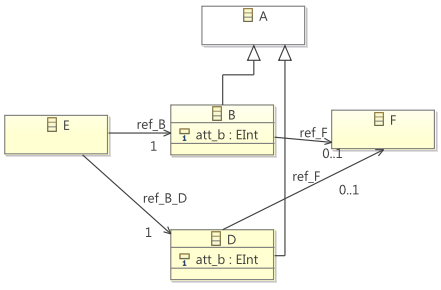


Fig. 6. graphical view of metamodel target MMt3

Changing operators

- *newName* for a named elementl: by defining a different name *newName* (resp. *newURIName* for EPackage), the concept is renamed by using *newName*;
- *bound* for an EStructuralFeature: by defining a different upper *newUpper*, resp. lower *newLower* bound, the target bound is modified;
- *changeContainment* for an EReference ER: if *true*, the containment property of ER in the target is changed to its opposite.
- *changeAbstract* for an EClass EC: if *true*, the abstract property of EC in the target is changed to its opposite.

Changing name is not a neutral operation with regard to the correctness of the metamodels produced. For EPackages, we do not accept that two transformed EPackages have the same name. Two EClasses EC1 and EC2 may be transformed into the same EClass EC3. But, this operation is allowed only if the properties are the same for EC3 after both transformations from EC1 and from EC2.

2.3 Model migration

When target metamodel MMt is obtained from MMs and MMs2MMt, it is possible to generate model migration from MMs model to MMt model :

- each instance of not deleted and not abstract EClass is copied to an instance of corresponding concrete EClass target; the not removed EAttributes and EReferences are also copied;
- for bound modifications, the current implementation considers only two cases: from (0..1) or (1..1) to (0..n) or (1..n), and the contrary. For the first case, we add the single property, if it exists, to the set; for the second case, we keep only the first element of the set, if it exists;
- when adding an EReference in MMt, we explore the model source graph to build the correct EReference in the target model;
- a Root class has to be specified in MMs to define the starting point of the migration transformation.

Changing the EReference containment property is not neutral. Since, an instance has only one container, adding containment relationship can lead to an association loss. In the same way, changing an abstract property can lead to an instance loss. If an EClass becomes abstract, the instances are not copied.

During experiments, generated migration has to be manually adapted to integrate some domain knowledge. It happens in three cases :

- it is necessary to add some extra properties in MMt: because the proposed work does not support an *add* operator, we manually add some extra initialization operations for the concerned instances;
- different properties in MMs are necessary to build one property in MMt: because Modif considers one-to-one operations, we manually modify property initialization in the generated transformation;
- only some specific instances of source model are transformed: because Modif considers generic editing operators, we manually add evaluation on properties when adding a concept in the target model.

2.4 Modif tools

Modif model appears to be the key of the process. So, we provide some facilities to build it (see figure 1). An ATL [8] transformation generateNoModif.atl, resp. eraseAll.atl, allows to generate a by-default Modif model. GenerateNoModif generates a noModifMMs.modif: all the concepts are not removed and not modified, coupled with ModifMM.kmt, generates a copy of MMs. On the contrary, eraseAll generates an eraseMMs.modif: all the concepts are removed, coupled with ModifMM.kmt, generates an empty metamodel. At the model level, using the by-default noModifMMs model produces a model copier.

The Modif model designer may then use a by-default Modif model to build a specific one. These tools are particularly useful when manipulating large modeling language, such as UML, to build a first Modif model, easy to tune for specific needs.

3 Experiment

3.1 UML to FSM

To test our approach, for DSML design, we propose to develop a Finite State Machine metamodel called myFSM as an evolution of UML. UML.ecore is considered here as the source MMs, myFSM.ecore is considered as the target MMt. Then, we use an existing tool flattenFSM on myFSM model, which encapsulates the semantic of FSM by transforming a concurrent and hierarchical FSLM to a flattened one. The developed tools are illustrated on a model called Train.uml.

3.2 FSM part of UML

First we transform the UML model to a subset of UML corresponding to FSM part called umlFSM. The target metamodel is based on the UML documentation for FSM (chapter 15 of UML superstructure documentation). UML2umlFSM.modif is the result of the following process:

- first, by using eraseAll.atl, remove property is set to true for all concepts;
- then, for all the concepts that are present in the UML documentation for FSM, remove property is set to false;
- we also keep (remove property to false) some concepts which encapsulate some useful information for further analysis: Package, PackageableElement, Event, ReceiveOperationEvent and OpaqueBehavior;
- we *hide* concepts that are not present in the UML documentation but that have an inheritance relationship between concepts present in umlFSM : Type, Classifier, Class, BehavioredClassifier and MessageEvent.

The obtained umlFSM (see figure 7) is close to the UML documentation. From the UML.ecore used, we note some differences with the UML standard:

- there is no FinalState EClass;
- we add some extra information (Package, PackageableElement, Event, ReceiveOperationEvent and OpaqueBehavior);

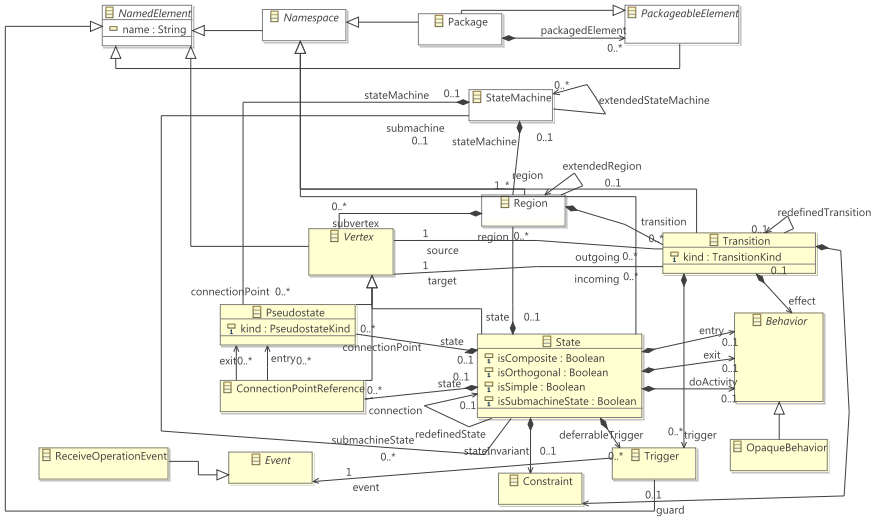


Fig. 7. graphical view of metamodel umlFSM.ecore

3.3 umlFSM to myFSM

From umlFSM, myFSM (see figure 8) metamodel is obtained through a chain of transformations (see figure 9). From umlFSM, we first produce an intermediate metamodel called iFSM :

- all information is hidden except for that present in myFSM (*package* is renamed as *Root*);
- *NamedElement* is *flattenAll* to keep *name* property for non hidden concepts;

Because myFSM is not exactly a subset of umlFSM, we need to manually modify iFSM to produce iaFSM and to adapt the corresponding generated migration:

- both UML *state* and UML *Pseudostate* become a state in myFSM; we add an extra Boolean attribute called *ini* to differentiate the initial state from others; set to *true* for UML Pseudostate only if its *kind* is equal to *initial*;
- a string *effectName* attribute is added to Transition, concatenating guard, trigger and effect names;

Then from iaFSM, we build myFSM by renaming and removing some unnecessary EReferences, added by the first transformation. From the corresponding generated and adapted migrations, we generate the model and flatten it using a specific tool flattenFSM.

Only a few days were necessary to develop the models and the migrations, including the flattenFSM tool. Extra user code is limited and was easy to insert in the generated transformation.

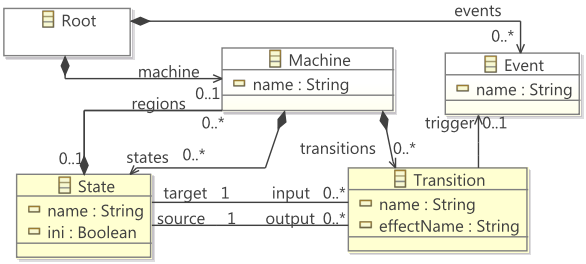


Fig. 8. graphical view of metamodel myFSM.ecore

3.4 Application

We illustrate the toolchain with a UML model of a toy train. Pushing a button which enables to turn the train on or off. For the first time, only the lights are turned on and the train stands still. The second time the button is pushed, the lights are turned on and the train moves forward. The next time, only the lights are turned on. And finally, the lights are turned on and the train moves backward. Then the same cycle of behavior begins again. This behavior is formally stated in UML as depicted in figure 10. In the UML model, the designer separated two sub-behaviors: one for the lights (on or off) , and one for the movement (stop, forward and backward). The event named *click* stands for the pushing button action.

This hierarchical state machine has to be flattened before being analyzed. After migration to myFSM model and using specific tools, the FSM model train is shown in figure 11.

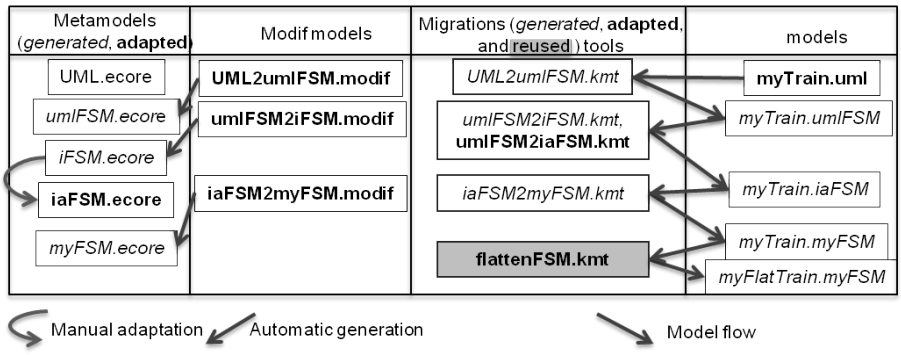


Fig. 9. metamodels, modif models transformation and models from UML to myFSM

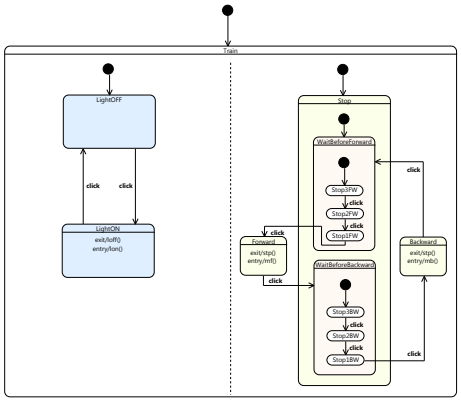


Fig. 10. UML graphical view of train model behavior

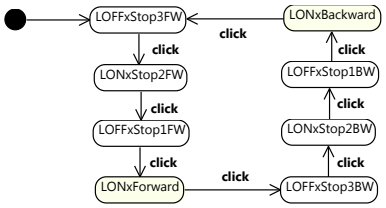


Fig. 11. Flattened version of FSM for mode ofl train

4 Related works

Recently, many works propose approaches to support metamodel evolutions and model migration generation [13]. For automatic migration generation, literature exhibits two strategies: first, ones [5] [7] [9] [16] that are based on editing (add, remove and change) and refactoring operators, and others [4] [1] [14] [2] , on explicit matching relationships between concepts. The goal of Modif is to help metamodel design from an existing metamodel, so we follow the first strategy, by encapsulating editing and refactoring operators. Compared to literature, the main originality of Modif is the *hide* operator, a weak implementation of *remove* operator, keeping implicit relationships between concepts; and also *flatten* implementation, keeping all references at model level. Other works concentrate on model migration and metamodel evolution semantic [10] [6] [15]. These works are complementary to ours, to evaluate the correctness of transformations, and will be investigated later.

5 Conclusion

Modif model helps DSML design and migration generation. Modif is based on classical refactoring (flatten), editing operators (remove, rename, change) and original ones (hide, flattenAll) defined at metamodel level. Experiments on developing a UML2FSM transformation tool show the efficiency of the approach to develop metamodels and migrations quickly, concentrating on the necessary information, that is to say, metamodel differences. In perspective, we are working on more complex operators (refactoring based on pattern transformation: list to set ...), java code generation instead of Kermeta and inversible transformations, to provide tools to go back to the source model, after target model generation and modification.

References

1. Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 222–231, Washington, USA, 2008. IEEE Computer Society.
2. Marcos Didonet Del Fabro and Patrick Valduriez. Towards the efficient development of model transformations using model weaving and matching transformations. *Software and Systems Modeling*, 8(3):305–324, July 2009.
3. Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf.
4. Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. Managing model adaptation by precise detection of metamodel changes. In *ECMDA-FA*, pages 34–49, 2009.
5. Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. Cope - automating coupled evolution of metamodels and models. In *ECOOP 2009*, Genoa, pages 52–76. Springer-Verlag, 2009.
6. Markus Herrmannsdoerfer and Maximilian Koegel. Towards semantics-preserving model migration. In *international MoDELS Workshops on Model Evolution*, pages 33–42, 2010.
7. Markus Herrmannsdoerfer, Sander D. Vermolen, and Guido Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *3rd International Conference on Software Language Engineering*, Eindhoven, pages 163–182. Springer-Verlag, 2010.
8. Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *MoDELS Satellite Events*, pages 128–138, 2005.
9. Stefan Jurack and Florian Mantz. Towards metamodel evolution of emf models with henshin. In *international MoDELS Workshops on Model Evolution*, pages 90–95, 2010.
10. Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A manifesto for semantic model differencing. In *international MoDELS Workshops on Model Evolution*, pages 73–82, 2010.
11. Muller, Pierre-Alain and Fleurey, Franck and Jézéquel, Jean-Marc. Weaving executability into object-oriented meta-languages, October 2005.
12. Object Management Group. *UML Infrastructure specification,* `http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/`. 2007.
13. Louis Rose, Markus Herrmannsdoerfer, James Williams, Dimitrios Kolovos, Kelly Garcés, Richard Paige, and Fiona Polack. A Comparison of Model Migration Tools. In *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 61–75, Berlin, Heidelberg, 2010.
14. Jonathan Sprinkle and Gabor Karsai. A domain-specific visual language for domain model evolution. *J. Vis. Lang. Comput.*, 15(3-4):291–307, 2004.
15. Perdita Stevens. Bidirectional model transformations in qvt: Semantic issues and open questions. In *MoDELS*, pages 1–15, 2007.
16. Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In *ECOOP*, pages 600–624, 2007.

Summarizing Semantic Model Differences

Shahar Maoz*, Jan Oliver Ringert**, and Bernhard Rumpe

Software Engineering
RWTH Aachen University, Germany
http://www.se-rwth.de/

Abstract. Fundamental building blocks for managing and understanding software evolution in the context of model-driven engineering are differencing operators one can use for model comparisons. Semantic model differencing deals with the definition and computation of semantic diff operators for model comparison, operators whose input consists of two models and whose output is a set of diff witnesses, instances of one model that are not instances of the other. However, in many cases the complete set of diff witnesses is too large to be efficiently computed and effectively presented. Moreover, many of the witnesses are very similar and hence not interesting. Thus, an important challenge of semantic differencing relates to witness selection and presentation.
In this paper we propose to address this challenge using a summarization technique, based on a notion of equivalence that partitions the set of diff witnesses. The result of the computation is a summary set, consisting of a single representative witness from each equivalence class. We demonstrate our ideas using two concrete diff operators, for class diagrams and for activity diagrams, where the computation of the summary set is efficient and does not require the enumeration of all witnesses.

1 Introduction

Differencing operators used for model comparisons are fundamental building blocks for managing and understanding software evolution in model-driven engineering. Semantic model differencing [12] deals with the definition and computation of semantic diff operators, whose input consists of two models, e.g., two versions along the history of a model, and whose output is a set of diff witnesses, instances of one model that are not instances of the other. Each witness serves as a concrete proof for the difference between the two models and its meaning.

However, the complete set of diff witnesses is in many cases too large to be efficiently computed and effectively presented. Moreover, many of the witnesses are very similar and hence not interesting. Thus, an important challenge of semantic differencing relates to witness computation, selection, and presentation.

In this paper we propose to address this challenge using the definition of a summarization technique, based on a notion of equivalence that partitions the

* S. Maoz acknowledges support from a postdoctoral Minerva Fellowship, funded by the German Federal Ministry for Education and Research.
** J.O. Ringert is supported by the DFG GK/1298 AlgoSyn.

set of diff witnesses. The result of the summarization is a summary set, consisting of a single representative witness from each equivalence class.

In recent work we have presented two concrete semantic diff operators, *cddiff* [11] for class diagrams (CDs) and *addiff* [9] for activity diagrams (ADs), along with the algorithms to compute them and with an initial evaluation of their performance and the usefulness of their results. Here we demonstrate the application of the summarization technique to these two concrete diff operators. Moreover, the computation of the summary set is efficient and does not require the enumeration of all witnesses.

It is important to note that we do not look for a difference summary in the form of a succinct mathematical representation of all differences between the two models, e.g., in the case of activity diagrams, a state machine accepting exactly all those traces accepted by one model and not by the other. Rather, we believe that in order to make semantic differencing useful and attractive to engineers, one needs to take a constructive and concrete approach: to compute and present concrete, specific, and thus easy to understand witnesses for the difference (e.g., in the case of activity diagrams, concrete execution traces).

Sect. 2 presents examples to motivate the need for summarization. Sect. 3 presents a formal, language independent overview of our approach and continues with its specializations for CDs and ADs. Sect. 4 briefly describes the algorithms used to compute the summarized sets of witnesses. Initial evaluation and discussion appear in Sect. 5. Related work is discussed in Sect. 6 and Sect. 7 concludes.

2 Examples

Example I. Consider $cd.v1$ of Fig. 1, describing a first version of a model for (part of) a company structure with employees, managers, and tasks. A design review with a domain expert has revealed three bugs in this model: (1) the number of tasks per employee should not be limited to two; (2) managers are also employees, and they can handle tasks too; (3) an employee must have exactly one manager. These bugs have been addressed in the second version $cd.v2$.

Diff witnesses for the semantic difference between $cd.v2$ and $cd.v1$ are object models that are in the semantics of $cd.v2$ and not in the semantics of $cd.v1$. Fig. 2 shows two such diff witnesses: om_1, consisting of an employee with three tasks, who is managed by a manager; and om_2, consisting of a manager that manages herself, without any tasks. However, these are only examples. Many more diff witnesses exist, e.g., those that are similar to om_1 but include additional tasks, or those that consist of duplicates of om_1 and/or om_2 etc.

Example II. Consider the ADs of Fig. 3, describing three versions of a ticket reservation process. Witnesses for the semantic difference between two ADs are execution traces that are allowed by one AD and are not allowed by the other.

For example, traces of $ad.v2$ that are not in $ad.v1$ include (1) a trace with $tickets = 3$ where the action `accounts` comes before the action `reserve`, and (2) a trace with $tickets = 10$ (where $ad.v2$ executes actions `register` and `welcome msg`). Traces of $ad.v3$ that are not in $ad.v2$ are all traces with $tickets < 12$.

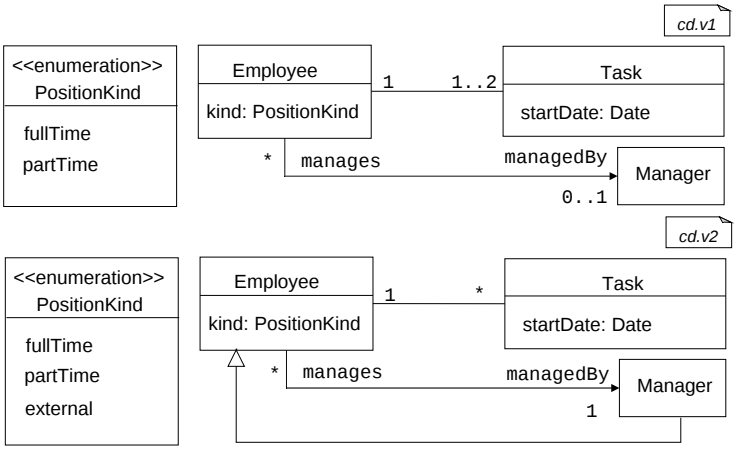


Fig. 1. Two versions of a CD, $cd.v1$ and $cd.v2$.

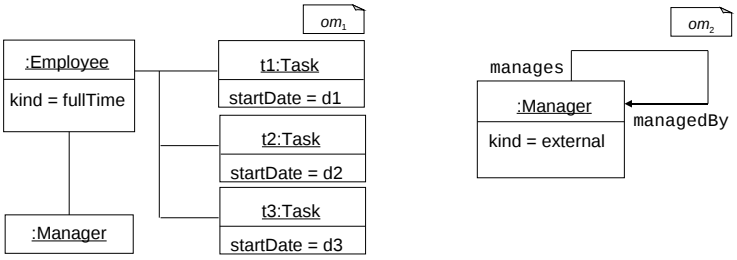


Fig. 2. Two diff witnesses from $cddiff(cd.v2, cd.v1)$.

Overall, there are many diff traces, due to the possible values of the input *tickets* and the partial order between `reserve`, `accounts`, and `updates`.

Given the large number of diff witnesses, for both *cddiff* and *addiff*, the challenge we address in this paper relates to the computation, selection, and presentation of a summarized set of witnesses.

3 Definitions

Consider a modeling language $ML = \langle Syn, Sem, sem \rangle$ where Syn is the set of all syntactically correct expressions (models) according to some syntax definition, Sem is a semantic domain, and $sem : Syn \to \mathcal{P}(Sem)$ is a function mapping each expression $e \in Syn$ to a set of elements from Sem (see [5]).

The semantic diff operator $diff : Syn \times Syn \to \mathcal{P}(Sem)$ maps two syntactically correct expressions e_1 and e_2 to the (possibly infinite) set of all $s \in Sem$ that are in the semantics of e_1 and not in the semantics of e_2. Formally:

Definition 1. $diff(e_1, e_2) = \{s \in Sem |\ s \in sem(e_1) \wedge s \notin sem(e_2)\}$.

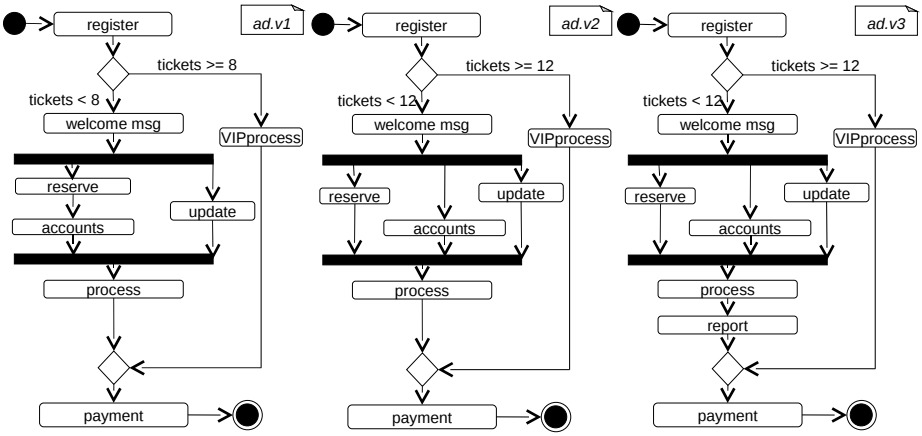


Fig. 3. Three versions of an AD for a ticket reservation process. The input variable *tickets* ranges from 0 to 15.

The elements in $\mathit{diff}(e_1, e_2)$ are called *diff witnesses*. When e_1 and e_2 are fixed, we use diff for the set $\mathit{diff}(e_1, e_2)$.

Let $Q = \{Q_1, Q_2, \ldots\}$ be a partition of diff, that is, $\mathit{diff} = \bigcup Q_i$, $\forall i : Q_i \neq \emptyset$ and $\forall i \neq j : Q_i \cap Q_j = \emptyset$. We define a partition function $\mathit{part} : \mathit{diff} \to Q$, which maps every diff witness $dw \in \mathit{diff}$ to an element Q_i of the partition Q such that $dw \in Q_i$. Note that Q, diff, and part all depend on fixed e_1 and e_2.

A summary of the set diff according to a partition Q, diff_Q, is a subset of diff consisting of a representative diff witness from each element in Q. Formally:

Definition 2. *Given a set of diff witnesses $\mathit{diff} = \mathit{diff}(e_1, e_2)$ and a partition Q, a summary of diff is a set $\mathit{diff}_Q \subseteq \mathit{diff}$ s.t.*

1. $\forall dw_1, dw_2 \in \mathit{diff}_Q$, $dw_1 \neq dw_2 \Rightarrow \mathit{part}(dw_1) \neq \mathit{part}(dw_2)$
2. $\forall Q_i \in Q$, $\exists dw \in \mathit{diff}_Q$ *s.t.* $\mathit{part}(dw) = Q_i$.

3.1 Specialization for class diagrams

In previous work we have defined *cddiff*, a specializations of *diff* for CDs [11]. We now present a related specialization of diff_Q.

Our semantics of CDs is based on [4] and is given in terms of sets of objects and their relationships. Thus, the elements of *cddiff* are object models (and they are presented to the engineer using object diagrams). To define cddiff_Q, we define a partition of the set of all object models $\mathit{cddiff} \subseteq OM$ into equivalence classes based on the set of classes instantiated in each object model. More formally:

Definition 3 (class-equivalent partition for object models). *The class-equivalent partition maps every object model $om_1 \in \mathit{cddiff}$ to the set of all object models in cddiff whose set of instantiated classes is equal to the set of classes instantiated in om_1: $\mathit{part}(om_1) = \{om |\ \mathit{classes}(om) = \mathit{classes}(om_1)\}$.*

For example, consider the CDs shown in Sect. 2. A summary of the semantic difference $cddiff(cd.v2, cd.v1)$, according to the class equivalence partition, will include exactly four object models: one consisting only of managers (an example representative is a single manager managing herself, with no tasks and no employees that are not managers), another one consisting of only managers and employees (an example representative is a manager who manages an employee with no tasks), another one with only managers and tasks (an example representative is a manager managing herself and having several tasks), and, finally, one consisting of managers, employees, and tasks (an example representative is a model consisting of a manager managing an employee with several tasks).

3.2 Specialization for activity diagrams

In previous work we have defined *addiff*, a specializations of *diff* for ADs [9]. We now present related specializations of $diff_Q$.

We use UML2 Activity Diagrams for the syntax of our ADs. In addition to action nodes, pseudo nodes (fork, decision, etc.), the language includes input and local variables (over finite domains), transition guards, and assignments. Roughly, the semantics of an AD is made of a set of finite action traces starting from an initial node, considering interleaving execution of fork branches, the guards on decision nodes etc. (a formal and complete semantics of our ADs is given in [10]). The elements of *addiff* are execution traces of one AD that are not possible in the other AD; we call them diff traces. Note that we do not require that the traces end in a final node; the diff trace stops as soon as one AD reaches an action that cannot be matched by an action in the other AD. The set *addiff* does not include traces that have a prefix that is by itself a diff trace. In addition, in [9] we limit the results to one shortest diff trace per initial state.

Diff traces can be considered a special kind of model-based traces [8]. Each diff trace is presented to the engineer both textually and visually, by enumerating and highlighting the nodes participating in the trace on top of the concrete syntax of the input ADs themselves (see [9]).

To define a summary of *addiff*, we define Q_l, which partitions the set of all diff traces based on the list of actions (action names) appearing in each trace; i.e., traces that differ in terms of input or internal variable values but agree on the list of actions to be executed are considered equivalent. More formally:

Definition 4 (action-list-equivalence partition for traces). *The action-list-equivalent partition maps every trace tr_1 to the set of all traces whose list of executed actions is equal to the list of executed actions in tr_1: $part(tr_1) = \{tr|\ tr|_{action} = tr_1|_{action}\}$.*

For example, considering the ADs shown in Sect. 2, $addiff_{Q_l}$ of traces in $ad.v2$ that are not possible in $ad.v1$ will include (1) a trace with $tickets < 8$ where `accounts` comes immediately after `welcome msg`, and (2) one trace with $8 \leq tickets < 12$ ending with `welcome msg`. That is, the summary will include only 2 traces, each consisting of a different list of actions. However, $addiff_{Q_l}$

of traces in $ad.v3$ that are not possible in $ad.v2$ will include 6 traces, all with $tickets < 12$ and ending with the action `report`, due to the 6 possible orderings of the actions inside the fork/join (`reserve`, `accounts`, `update`).

Thus, we suggest also an alternative partition Q_s, where two traces are considered equivalent iff the sets of actions included in them are identical. This induces a coarser partition, as it abstracts away the order of actions in the traces. More formally:

Definition 5 (action-set-equivalence partition for traces). *The action-set-equivalent partition maps every trace tr_1 to the set of all traces whose set of executed actions is equal to the set of executed actions in tr_1: $part(tr_1) = \{tr |\ actions(tr) = actions(tr_1)\}$.*

Applying this coarser partition to our example, the summary for traces in $ad.v3$ that are not in $ad.v2$ includes only a single trace, where $tickets < 12$.

Finally, it is important to note that in addition to the concrete representatives, as part of the results of the computation for $addiff_{Q_l}$ and $addiff_{Q_s}$ (see below), we have symbolic representations of the initial states related to each equivalence class. These can be presented to the engineer together with the concrete traces, as part of the summary.

4 Computing Summaries

A naive approach to compute $diff_Q$ would first compute and enumerate all diff witnesses in *diff* and then group them into equivalence classes according to the given partition and choose one witness from each class. This approach, however, is inefficient, as the total number of witnesses is typically an order of magnitude larger than the number of equivalence classes in the partition. Thus, a more efficient approach should be taken. We give an overview of our approach to compute $diff_Q$, for *cddiff* and *addiff*, given the partitions suggested above.

4.1 Computing summaries for *cddiff*

In [11] we showed how *cddiff* can be computed (in a bounded, user-defined scope) using a translation to Alloy. Roughly, the translation takes two CDs as input and outputs an Alloy module whose instances, if any, represent object models in the semantics of one CD that are not in the semantics of the other. Computing another witness is done by asking Alloy for another instance of the module (technically, by constraining the SAT solver further to not allow the instances that were already found).

To compute $cddiff_Q$, when a diff witness is found, rather than simply asking Alloy for another witness, we generalize the instance that was found to its set of classes, and create a new predicate that specifies that it should not be the case that this set of classes consists of exactly the classes appearing in an instance of the Alloy module. We then rerun Alloy on a revised module, strengthened by the new predicate. This guarantees that a new instance, if any is found, would

be a diff witness from a different equivalence class. We iterate until no more new diff witnesses are found.

The above technique is guaranteed to provide a single representative from each equivalence class without the need to enumerate all witnesses first. However, like all other analysis done with Alloy, it is bounded by a user-defined scope. Also, its performance may not scale well for large CDs. Addressing these limitations may require the use of a completely different solution, i.e., not using Alloy, and is left for future work.

4.2 Computing summaries for *addiff*

In [9] we showed how *addiff* can be efficiently computed using a symbolic fixpoint algorithm, based on BDDs and the technologies of symbolic model-checking [2]. The algorithm starts with a representation of all non-corresponding states. It then moves 'backward', and adds to the current set of states, states from which there exists a successor in one AD such that for all successors in the other AD, the resulting successor pair is in the current set of states. The steps 'backward' continue until reaching a least fixpoint, i.e., until no more states are added. When the fixpoint is reached, the algorithm checks whether the fixpoint set includes initial states. For each such initial state, if any, the algorithm uses the sets of states computed during the backward steps to move forward (from the minimal position it can start from) and construct shortest diff traces.

To compute $addiff_{Q_l}$ we start with the first phase of the original algorithm and symbolically compute the set of all initial states from which a diff trace may start and all sets of states included in all diff traces. Then, rather then enumerating all concrete diff traces by computing a concrete diff trace starting in each initial state, we start with the set of all initial states and symbolically move forward to the set of all next states. If two or more actions are possible in the next step, we split the set of next states according to their action and continue, symbolically, for each of the sets in the split. We iterate this until reaching the differentiating actions, i.e., until no corresponding next state exists. Finally, for each symbolic trace we now have, we choose one initial state and compute a concrete trace that starts from it. We symbolically represent the set of initial states that share the list of actions in the trace (e.g., with ranges of input variables).

Computing a summary with our coarser partition, $addiff_{Q_s}$, is similar. When we are done with computing the symbolic traces of the action-list partition, before choosing concrete representatives, we iterate over the set of symbolic traces and eliminate any symbolic trace whose set of actions already appeared (in another order) in a previous trace. For each of the remaining symbolic traces, we choose one initial state and compute a concrete diff trace that starts from it.

5 Initial Evaluation and Discussion

We have applied the above summarization strategies for *cddiff* and *addiff* to the examples of Sect. 2. Table 1 lists the results in terms of the number of diff witnesses (object models, traces) found, with and without summarization.

For *cddiff*, all our examples have 20 or more diff witnesses without summarization (we computed *cddiff* with a scope of 10 and stopped after finding 20 witnesses). The number of witnesses found with summarization was only 3 or 4. The results show the effectiveness of the summarization approach in significantly reducing the number of diff witnesses presented to the engineer while keeping the set as diverse as possible. Also, note that finding only 3 witnesses means that the SAT solver was executed only 4 times (in the last execution, no diff witness was found). This shows the efficiency of our approach.

For *addiff*, the number of diff traces found without summarization varied: for some examples there are only few diff traces, while for others the number of diff traces found was much higher, up to 72. Applying summarization to the examples with a small number of witnesses does not make much difference. However, applying summarization to the examples with the many witnesses results in significantly smaller sets of witnesses, up to at most 6 representative traces for each example. For the action-list partition, significant reduction is observed when the ADs state space is large due to many possible inputs (many variables or variables with large domains like our *tickets* variable). For the action-set partition, further reduction is observed when the ADs' state space is large and where differences occur after some fork/join blocks with much partial order.

In the general case, summarization may entail information loss: one cannot always use the summary to enumerate all witnesses. Yet, in some cases, it is possible to keep an efficient symbolic representation of each equivalence class within the summary, so that *diff* can be easily computed from diff_Q. For example, the computation of addiff_{Q_l}, based on the action-list partition, includes a symbolic representation of all input states where diff traces may start. Given initial states and the list of actions that characterize each of the partitions, all diff traces can be reconstructed. For addiff_{Q_s}, based on the action-set partition, however, this is not the case; once the order of actions is abstracted away, one cannot use an initial state to generate a trace that is guaranteed to be a diff trace.

Finally, we consider the following alternatives for semantic differencing summarization. First, one may suggest a partition based on syntactic differences,

Name	# Wit. found	# Wit. found with summarization
*cd.v*1 vs. *cd.v*2	20	3
*cd.v*2 vs. *cd.v*1	20	4
*ad.v*1 vs. *ad.v*2	4	1/1
*ad.v*2 vs. *ad.v*1	20	3/3
*ad.v*2 vs. *ad.v*3	72	6/1
*ad.v*3 vs. *ad.v*2	72	6/1
*ad.v*1 vs. *ad.v*3	28	4/2
*ad.v*3 vs. *ad.v*1	36	5/4

Table 1. Results of applying the summarization strategies to the examples from Sect. 2. We computed CDDiff with scope 10 and stopped after 20 witnesses were found. For ADDiff summarization we show the number of witnesses according to the action-list partition / action-set partition.

i.e., such that witnesses are classified according to the syntactic differences they 'cover'. Second, in addition to partitioning, one may be interested in defining a (partial) order, such that the summarization method chooses a representative that is also minimal within its equivalence class. For example, in the case of *cddiff*, a partial order may be defined based on diff witness size, i.e., the number of objects in the object model. In the case of *addiff*, a partial order may be defined based on diff traces length. It seems that smaller diff witnesses would be easier to present and understand. More generally, rather than ordering witnesses locally, within each equivalence class, one may suggest a pre-order on all diff witnesses and look for global minimal ones.

Formalizing and evaluating these alternatives is left for future work.

6 Related Work

The problem of summarizing semantic differences is close to the problem of effective design space exploration [7], where the goal is to quickly visit a diverse set of solutions across a design space. The approach in [7] takes a user-defined notion of equivalence as input, and generates symmetry breaking predicates, which ensure that the underlying exploration engine does not sample multiple equivalent design candidates. In addition, the work employs randomization to incrementally construct a diverse set of non-isomorphic solutions, ideally making the solver 'jump around' various parts of the design space, sampling a wide variety of solutions. The work is integrated in a tool called FORMULA, which uses an SMT solver.

In [6], the authors present a technique to summarize all counterexamples of an LTL model-checking problem. They generalize concrete examples found by the SMV model-checker into equivalence classes, describe these with LTL formulas, and re-run the model-checker on a revised formula where examples that are equivalent to previously found ones are not considered. The work suggests four specific kinds of equivalence, at different levels of abstraction.

Our work is similar, in that we use class equivalence as a criteria for results selection and presentation. It is also very different, as it is specifically applied to the problem of semantic differencing for several modeling languages, and thus the criteria for 'symmetry breaking' and the technologies used (Alloy/SAT, BDD-based algorithms) are specific and very different than the ones in [6, 7].

Many works present syntactic approaches to differencing (e.g., [1, 13, 14]). Some related tools support hierarchical presentation of differences where the hierarchy is defined by the abstract syntax tree (AST) [3]. All differences are computed but the presentation in the AST can encapsulate them under collapsed sub-trees. This may be viewed as a form of presentation summarization. Note that our summarization technique is not limited to the presentation but is applied already as part of the computation: we show how to compute the summary set without the enumeration of all witnesses during the computation.

We are not aware of other work in the domain of software evolution that is directly related to differences summarization.

7 Conclusion

We have presented summarization techniques for semantic model differencing. We motivated the challenge of summarization and suggested ways to address it in the context of CD and AD semantic differencing. We demonstrated the utility of our summarization approach in providing a small yet informative set of diff witnesses and discussed alternatives and future challenges.

Future work includes the integration of the techniques presented here into the prototype implementations presented in [9] and [11]. Moreover, as we extend semantic differencing to additional languages, e.g., feature models and statecharts, we will be looking for summarization techniques for these languages too.

References

1. M. Alanen and I. Porres. Difference and Union of Models. In *Proc. 6th Int. Conf. on the UML*, volume 2863 of *LNCS*, pages 2–17. Springer, 2003.
2. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.
3. EMF Compare. http://www.eclipse.org/modeling/emft/?project=compare.
4. A. Evans, R. B. France, K. Lano, and B. Rumpe. The UML as a Formal Modeling Notation. In J. Bézivin and P.-A. Muller, editors, *Proc. UML*, volume 1618 of *LNCS*, pages 336–348. Springer, 1998.
5. D. Harel and B. Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, 2004.
6. A. L. Juarez-Dominguez and N. A. Day. On-the-fly counterexample abstraction for model checking invariants. Technical Report CS-2010-11, School of Computer Science University of Waterloo, Canada, 2010.
7. E. Kang, E. K. Jackson, and W. Schulte. An approach for effective design space exploration. In R. Calinescu and E. K. Jackson, editors, *Monterey Workshop*, volume 6662 of *LNCS*, pages 33–54. Springer, 2010.
8. S. Maoz. Model-based traces. In M. R. V. Chaudron, editor, *MoDELS Workshops*, volume 5421 of *LNCS*, pages 109–119. Springer, 2008.
9. S. Maoz, J. O. Ringert, and B. Rumpe. ADDiff: semantic differencing for activity diagrams. In *ESEC / SIGSOFT FSE*, pages 179–189. ACM, 2011.
10. S. Maoz, J. O. Ringert, and B. Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB 2011-07, RWTH Aachen University, Germany, 2011.
11. S. Maoz, J. O. Ringert, and B. Rumpe. CDDiff: Semantic differencing for class diagrams. In *ECOOP*, volume 6813 of *LNCS*, pages 230–254. Springer, 2011.
12. S. Maoz, J. O. Ringert, and B. Rumpe. A manifesto for semantic model differencing. In J. Dingel and A. Solberg, editors, *MODELS 2010 Workshops*, volume 6627 of *LNCS*, pages 194–203. Springer, 2011.
13. D. Ohst, M. Welle, and U. Kelter. Differences between versions of UML diagrams. In *Proc. ESEC / SIGSOFT FSE*, pages 227–236. ACM, 2003.
14. Z. Xing and E. Stroulia. Differencing logical UML models. *Autom. Softw. Eng.*, 14(2):215–259, 2007.

On the Use of Operators for the Co-Evolution of Metamodels and Transformations

Steffen Kruse

OFFIS - Institute for Information Technology
Escherweg 2, 26121 Oldenburg, Germany
steffen.kruse@offis.de
http//www.OFFIS.de

Abstract. The artefacts used in model driven approaches are often tightly coupled. Besides models being bound to metamodels by a conformance relation, transformation descriptions are defined on metamodels to perform their function. When the metamodels are changed during development or due to changing requirements, existing transformations need to be adapted. We propose a set of operators to ease this task. The operators are applied to metamodels to perform a change and allow the (semi-) automatic co-evolution of transformations.

Keywords: metamodel, model transformation, co-evolution, operators

1 Introduction

Model Driven Software Development (MDSD) is an approach to software engineering to ease the handling of complexity in the development and maintenance of modern software systems. MDSD is based on the concept of a model as a representation of a (software) system [2]. Models are expressed in languages, models of which in turn are called metamodels. Further techniques used in MDSD are code generation (where source code is generated from a set of models) and model to model transformation, of which code generation can be seen as a special case. The artefacts used in MDSD projects are tightly coupled. Models must be conformant to their respective metamodel to be valid as model transformations must match one or more metamodels to perform their function. This can become a problem when faced with evolution: as metamodels are extended or adapted, the dependencies may break until all dependent artefacts are adapted accordingly. The problem of co-evolution for metamodels and models has been addressed by a number of approaches (see for example [3, 8–10]). While being related, the problem of the co-evolution of metamodels and model transformations poses different and unique problems.

In this paper, we propose an initial set of operators that can be applied on a given metamodel to perform changes and allow the automatic or semi-automatic co-evolution of transformation descriptions. Atomic operators can be combined to form more complex ones. We implemented these operators in Java and tested them on copy transformations described in ATL to gain initial insight into the viability of this approach.

2 Impact of metamodel changes on transformations

Model to model transformations play an important role in model driven software development (MDSD) [16]. Numerous languages have been developed, dedicated solely to specify such transformations [4]. We chose the declarative parts of the Atlas Transformation Language (ATL) [11] as a starting point for our operator set. In essence, a model transformation defined in ATL is made up of a set of rules, where each rule produces one or more target model elements from a source model element. The application of a rule is determined by a source pattern, defined in terms of a source metamodel. The created elements are conformant to a target metamodel. ATL relies on a slightly adapted version of the Object Constraint Language (OCL) to perform model navigation and to calculate values for target properties. To analyse the impact on metamodel changes on transformation rules in ATL, we first discern which metamodel involved in the transformation is changed. We refer to the source metamodel as the left-hand-side metamodel (LHS) and the target metamodel as right-hand-side (RHS) respectively.

Figure 1 shows an excerpt of the (informal) syntax of an ATL Matched Rule, taken from the ATL Language Guide [1]. Metamodel changes to the LHS can impact the following parts of an ATL rule: ① The source pattern ② conditions imposed on the source pattern (expressed in OCL) and ④ the binding assignment. Changes to the RHS are potentially reflected in: ③ the target pattern (each rule can have numerous target patterns) and ④ the definition of properties in the binding. We leave out the imperative part ((the *do*-statement)) and the variable section (see the ATL User Guide [1] for further detail). Both are to be looked at in the future, building on the work presented here.

```
rule rule_name {
    from
        in_var : in_type [in model_name]? [(
            condition
        )]?
    [using {
        var1 : var_type1 = init_exp1;
        ...
        varn : var_typen = init_expn;
    }]?
    to
        out_var1 : out_type1 [in model_name]? (
            bindings1
        ),
        out_var2 : distinct out_type2 foreach(e in collection)(
            bindings2
        ),
        ...
        out_varn : out_typen [in model_name]? (
            bindingsn
        )
    [do {
        statements
    }]?
}
```

Fig. 1. ATL MatchedRule Syntax

OCL is used for two purposes in ATL: to formulate conditions on the applicability of rules and to calculate values in the property bindings. In both cases, OCL statements may navigate over models and can potentially involve any part of the LHS metamodel. This can lead to very complex source patterns for complex rules. We simplify the influence of OCL expressions on the evolution process for a start, by either disabling the application of operators when they impact expressions or delegating to human intervention (bar rename operations which can be easily undertaken for OCL expressions). Depending on the operator and the expression structure, further adaption of the expressions may be possible, which would broaden the use of the operator. We leave this investigation for future work. Markovic and Baar have shown how OCL expressions can be migrated along with UML models [13], which gives good pointers to the adaptation of complex OCL expressions in transformation rules.

3 Operators

Table 1 summarizes the influence of the operators on transformation rules, depending on where they occur and which part of the rule they affect. For the LHS these are the source pattern element (Source P.), the condition, and the binding assignment (Binding A.). For the LHS they are the target pattern (Target P.) and the property binding (Binding). The kind of adaptation possible is given as follows: [*A*] stands for automatic adaptation, [-] the operator has no influence and [*H*] human intervention is needed. In cases where we list the two possibilities [*A/H*], automatic adaptation is possible most of the time – depending on further detail on the kind of change or the structure of the rule (see the detailed description of the operators.) The next sections discuss the operators and their influence in more detail.

3.1 Atomic Operators

- **Rename Class/Attribute/Relation**: Elements (classes/attributes/relations) of a metamodel are renamed. These operators are very similar – both in their pre-conditions and influence on transformation rules. Names of elements are required to be unique in their given namespace, so existing names are excluded by pre-condition when applying these operators on elements. For attributes and relations this also has to be checked for inheritance. When the pre-conditions hold, metamodel references can be adapted automatically in all parts of transformation rules.
- **Add Class**: A new class is added to the metamodel. The class must be unique in its namespace. The syntactic correctness is preserved for transformation rules on both sides, as existing rules are not influenced by new elements. Depending on the semantics of the transformation, new rules may have to be created by hand to provide for the new class.
- **Add Relation / Attribute**: A new relation or attribute (property) is added to a class in the metamodel. Its name must be unique for the owning class

Operator	LHS			RHS	
	Source P.	Condition	Binding A.	Target P.	Binding
Rename Class	A	A	A	A	A
Rename Attribute	-	A	A	-	A
Rename Relation	-	A	A	-	A
Add Class	A	A	A	A	A
Add Relation	-	A	A	-	A/H
Add Attribute	-	A	A	-	A/H
Delete Class	A	-	-	-	A
Delete Relation	-	A	A/H	-	A
Delete Attribute	-	A	A/H	-	A
Pull Up Attribute	-	A	A	-	A/H
Push Down Attribute	-	A	A/H	-	A
Pull Up Relation	-	A	A	-	A/H
Push Down Relation	-	A	A/H	-	A
Introduce Inheritance	-	A	A	-	A/H
Extract Superclass	A	A	A	A	A/H
Flatten Hierarchy	A	A	A	A	A

Table 1. Operator Influence

and classes along the inheritance hierarchy. For the LHS metamodel, transformation rules do not have to be adapted to preserver syntactic correctness. For the RHS, human intervention is required only when the added property is obligatory and no default value is supplied in the metamodel. In this case, all rules which feature a target pattern for the owning class or descending classes must be adapted to supply a value for the new property by hand.

– **Delete Class**: An empty and unreferenced class is deleted from the metamodel. To empty a class and remove all references to it first, the *Delete Property*, *Push Down Property* etc. operators can be applied prior. For the LHS, transformation rules that feature the deleted class as source pattern element can be removed. As the operator can only be used when the class is not referenced, the class cannot feature in the condition or binding assignment parts of rules for other source pattern elements. For the RHS, the target patterns for the class can be removed. Should this leave a rule without any target pattern, the entire rule can be removed.
– **Delete Relation / Attribute**: A relation or attribute (property) is deleted from a class. Rules that need to be adapted either refer to the class from which the property is removed or any subclass; or contain OCL expressions that refer to the removed property. The adaptation is as follows:
  • For the LHS:
    ∗ Conditions - If the deleted property is part of a condition restricting the application of a rule, the condition can be removed. This means that the rule potentially applies more often. The user should be warned of this behaviour.
    ∗ Binding Assignment - If the property is featured in a binding assignment of an RHS property, the whole assignment can be removed

if the RHS property is non-obligatory or has a default value. This would preserve syntactic correctness. Otherwise, human intervention is needed.
  - RHS: The property can have a binding for the target pattern of the class or the subclasses it was removed from. The binding assignment can be removed to preserve syntactic correctness.
- **Pull Up Relation / Attribute**: A relation / attribute (property) is pulled from all subclasses into a superclass and removed from the subclasses. This may mean that the user first has to introduce the property to subclasses that don't have the given property. This can be done by applying the *Add Reference /Attribute* operators.
  - Rules involving the superclass: All rules that match the superclass to which the property is pulled can be treated analogous to the *Add Relation/Attribute* operators. This means, for the LHS case, no change is needed, as the addition of a property does not break the syntactic correctness of the rules. For the RHS, human intervention is only needed if the property is obligatory and no default value is given. In all other cases no change is needed.
  - Rules involving the subclasses: All rules that match the subclasses from which the property is pulled require no adaptation, as the property remains available through inheritance.
- **Push Down Relation / Attribute**: A property is pushed from a superclass A into all subclasses that are removed from the superclass by exactly one level of inheritance. Subsequent application of the *Push Down Property* operator along the inheritance tree can push properties to further removed classes. Pushing properties to all child classes makes little sense on its own, it is expected that the user applies further operations like removing the property with the *Delete Property* operator on subclasses that don't use the property (this was the original intention of Fowlers Push Down Field Refactoring [7]) or removing the superclass with *Delete Class* if it contains no further properties.
  For this operator we have to discern between rules that use the superclass from which the property is removed and the subclasses that receive the property. For rules on the subclasses, the impact is simple: no adaption is needed as the property was previously available through inheritance. For the superclass, the behaviour is like the *Delete Relation / Attribute* operators:
  - For the LHS:
    * Conditions - If the removed property is part of a condition restricting the application of a rule, the condition can be removed. This means that the rule potentially applies more often. The user should be warned of this behaviour.
    * Binding Assignment - If the property is featured in a binding assignment of an RHS property, the whole assignment can be removed if the RHS property is non-obligatory or has a default value. This would preserve syntactic correctness. Otherwise, human intervention is needed.

  - RHS: The property can have a binding for the target pattern of the class it was pushed down from. The binding assignment can be removed to preserve syntactic correctness.
- **Introduce Inheritance**: Inheritance is introduced between two classes. For the LHS, no change is needed; the syntactic correctness of all rules is preserved. For the RHS, human intervention is only needed when the new superclass contains obligatory properties (without default values). Then all rules matching the new subclass must be adapted to supply a value for the newly inherited property. This is analogous to the *Add Attribute/Reference* operators.

3.2 Complex Operators

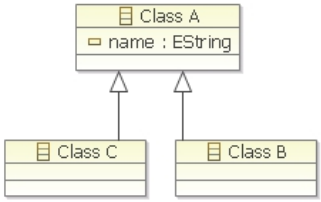
Complex operators are (in part) made up of simple operators. There are two reasons for providing complex operators: 1) convenience – to sum up common cases in one operator and 2) the use of the complex operator provides more information that can be used for the adaptation of transformations than just the combined use of simple operators. *Extract Superclass* is an example of the first case, while *Flatten Hierarchy* applies to both.

- **Extract Superclass**: A new superclass of a given class is introduced and some properties of the subclass are moved to the new superclass. This operator can be wholly made up of the Add Class, Introduce Inheritance and repeated use of Pull Up Property Operators. The same constraints and adaptions apply. This operator is very much like Fowlers *Extract Superclass* refactoring [7].

- **Flatten Hierarchy**: The purpose of this operator is to reduce the inheritance graph. At first glance, this operator can be implemented using the *Push Down Property* operator on all properties of the superclass until it is empty and then use the *Delete Class* operator to remove the superclass. Yet, a special case exists where this is impractical: Assume this operator is used on the LHS metamodel and there are rules that apply to the superclass in the source pattern. Before the change, the rule would be executed for all instances of subclasses as well as for those of the superclass. If the rule simply gets deleted (as with the Delete Class operator) target elements are no longer created for the subclass instances. We consider this to be an unexpected and inconvenient result for the user. Since we have more information available when the *Flatten Hierarchy* operator is used explicitly, we propose creating copies of rules that match the superclass for each subclass and changing the source pattern to match each subclass. Then the original rule for the superclass can be removed.

4 Evaluation

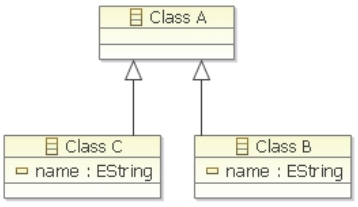
We implemented the operators in Java and perform them on metamodels in Ecore and the abstract syntax model of ATL transformations in the Eclipse




```
rule Class_A_COPY {
    from
        class_A : Metamodel!"Class A"
        (
            class_A.oclIsTypeOf(Metamodel!"Class A")
        )
    to
        class_A_COPY : Metamodel!"Class A" {
            name <- class_A.name
        )
}

rule Class_C_COPY {
    from
        class_C : Metamodel!"Class C"
        (
            class_C.oclIsTypeOf(Metamodel!"Class C")
        )
    to
        class_C_COPY : Metamodel!"Class C" {
            name <- class_C.name
        )
}
```

Fig. 2. Flatten Hierarchy: Initital State




```
rule Class_A_COPY {
    from
        class_A : Metamodel!"Class A"
        (
            class_A.oclIsTypeOf(Metamodel!"Class A")
        )
    to
        class_A_COPY : Metamodel!"Class A" {
        )
}

rule Class_C_COPY {
    from
        class_C : Metamodel!"Class C"
        (
            class_C.oclIsTypeOf(Metamodel!"Class C")
        )
    to
        class_C_COPY : Metamodel!"Class C" {
            name <- class_C.name
        )
}
```

Fig. 3. Flatten Hierarchy: after Push Down Attribute


```
rule Class_C_COPY {
    from
        class_C : Metamodel!"Class C"
        (
            class_C.oclIsTypeOf(Metamodel!"Class C")
        )
    to
        class_C_COPY : Metamodel!"Class C" {
            name <- class_C.name
        )
}
```

Fig. 4. Flatten Hierarchy: Final State

ATL infrastructure[1]. For an initial evaluation of the operators, we apply them to *copy transformations*. A copy transformation for a given metamodel produces an exact copy of all models conforming to the metamodel. It features the same metamodel both as source and target. Copy transformations can be generated by a higher-order transformation (HOT). This HOT takes any given metamodel and produces a matching copy transformation. Copy transformations can be used e.g. as foundations for model refinement [12]. To use operators for the evolution of copy transformations makes little sense in practice, as the transformation can be re-generated anytime the metamodel evolves. Yet, by comparing the two, we gain some initial insight into how the operators fare.

As an example, we discuss the use of the complex *Flatten Hierarchy* operator on a simple metamodel. We generated the copy transformation for the metamodel using Xpand of the Eclipse Model To Text (M2T) project[2]. The initial state of the metamodel and the corresponding copy transformation are shown in figure 2. (We have removed the copy rule for Class B for brevity in the figure; it is an exact duplicate of the rule for Class C.) The generated copy transformation contains a rule for each class, and the value of the *name* attribute is copied in the binding section. The condition in each source pattern ensures that the rule only matches exactly the given class and not subclasses (who have their own copy rules). We apply the *Flatten Hierarchy* operator to metamodel and transformation and interpret the change to have taken place on the RHS. The operator first uses the *Push Down Attribute* operator to move the *name* attribute from the superclass A to the subclasses B and C. As a result, the binding assignment for the attribute is removed for the copy rule of class A. The other rules are left unchanged (see figure 3). After the superclass A is relieved of its attribute, it can be removed. The result is shown in figure 4. In consequence, the copy rule for class A is removed from the copy transformation.

Generating the copy transformation for the new metamodel yields the same resulting set of rules. We take this as an indication that the approach is at least viable.

[1] see `http://eclipse.org/atl/`
[2] See `http://www.eclipse.org/modeling/m2t/?project=xpand`

4.1 Discussion

First results for copy transformations indicate that an operator based approach can be beneficial to the co-evolution problem for metamodels and model transformations. In many cases, at least syntactic correctness of transformations can be preserved when making changes. Yet, the semantics of transformations are very difficult to cater for. This is one of the main differences between the co-evolution transformations vs. co-evolution of models. The semantics of the conformance relation between metamodels and models is well defined, while the semantics of transformations very much depend on their purpose. For example, copy transformations have to regard every class and attribute of a metamodel. When a class is added to the metamodel, the rules of the copy transformation can be left unchanged for the transformation to be syntactically correct – yet it no longer performs according to its purpose.

Furthermore, the impact on transformations depends not only on the type of change performed on the metamodel, but also how the affected elements are used in transformation rules. It is possible that one transformation can be automatically adapted to a change in a metamodel, while another transformation using the same metamodel requires human intervention. We hope to find further insight in the future as to which kinds of transformations lend themselves well to automatic co-evolution and which don't.

5 Future and Related Work

Further operators need to be defined to make the set complete (any metamodel can be de- and reconstructed using operators.) Regarding the adaptation of OCL expressions along with the rules may broaden the use of operators. So far, we have only looked at transformations defined in ATL, but the given approach may lend itself well to other transformation languages and a comparison is seen to be beneficial. Finally, we plan to compare our operators to those used for metamodel and model co-evolution as a combined approach may make evolution easier – if it is possible.

The use of operators on co-evolution issues has been covered well for models [3, 8–10]. For metamodel and transformation co-evolution, Mndez et al. have described the problem and given first pointers to an operator-based approach [14]. Di Ruscio et al.[5] describe an initial approach using a DSL for both metamodel and model and transformation co-evolution. For OCL, Markovic and Baar have shown how OCL expressions can be migrated along with UML models [13]. Other approaches to the problem exist, for example based on evolution in the ontology space [15] or by always generating transformations for all metamodel versions [6]. The co-evolution problem is complex and has many facetts, as transformations have a vast field of application. Which approach is well suited for which case remains to be evaluated in the future.

References

1. Atl/user guide - the atl language, `http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language`, last seen: 14.07.2011
2. Bzivin, J.: In search of a basic principle for model driven engineering. In: UPGRADE. CEPIS (Council of European Professional Informatics Societies), NOVTICA (2004)
3. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. Enterprise Distributed Object Computing Conference, IEEE International 0, 222–231 (2008)
4. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. 45(3), 621–645 (2006)
5. Di Ruscio, D., Iovino, L., Pierantonio, A.: What is needed for managing co-evolution in MDE? In: Proceedings of the 2nd International Workshop on Model Comparison in Practice. pp. 30–38. IWMCP '11, ACM, New York, NY, USA (2011)
6. Didonet Del Fabro, M., Valduriez, P.: Towards the efficient development of model transformations using model weaving and matching transformations. Software and Systems Modeling 8, 305–324 (2009)
7. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (1999)
8. Garces, K., Jouault, F., Cointe, P., Bézivin, J.: Adaptation of Models to Evolving Metamodels. Research Report RR-6723, INRIA (2008)
9. Gruschko, B., Kolovos, D.S., Paige, R.F.: Towards synchronizing models with evolving metamodels. In: Proceedings of the Workshop on Model-Driven Software Evolution - MoDSE2007 at the 11th European Conference on Software Maintenance and Reengineering - CSMR 2007 (2007)
10. Herrmannsdoerfer, M., Vermolen, S.D., Wachsmuth, G.: An extensive catalog of operators for the coupled evolution of metamodels and models. In: Proceedings of the Third international conference on Software language engineering. pp. 163–182. SLE'10, Springer-Verlag, Berlin, Heidelberg (2011)
11. Jouault, F., Kurtev, I.: Transforming models with atl. In: Bruel, J.M. (ed.) MoDELS Satellite Events. Lecture Notes in Computer Science, vol. 3844, pp. 128–138. Springer (2005)
12. Kapova, L., Goldschmidt, T.: Automated feature model-based generation of refinement transformations. Software Engineering and Advanced Applications, Euromicro Conference 0, 141–148 (2009)
13. Markovic, S., Baar, T.: Refactoring ocl annotated uml class diagrams. Software and System Modeling 7(1), 25–47 (2008)
14. Méndez, D., Etien, A., Muller, A., Casallas, R.: Transformation migration after metamodel evolution. In: International Workshop on Model and Evolution. Oslo, Norway (October 2010)
15. Roser, S., Bauer, B.: Journal on data semantics xi. chap. Automatic Generation and Evolution of Model Transformations Using Ontology Engineering Space, pp. 32–64. Springer-Verlag, Berlin, Heidelberg (2008)
16. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Software 20, 42–45 (2003)

Towards Feature-Based Evolutionary Software Modeling

Hassan Gomaa

Department of Computer Science
George Mason University, Fairfax, VA
hgomaa@gmu.edu

Abstract. This paper proposes an evolutionary development approach, which uses software product line and feature modeling concepts for evolving multi-version systems. This model-based approach addresses both the original development and subsequent post-deployment evolution. The different versions of an evolutionary system are considered a software product line, with each version of the system a product line member. Evolution is built into the software development approach because variability in the software architecture is determined by considering the impact of each variable feature on the software architecture and evolving the architecture to address the feature. Being feature based, the approach closely relates the evolution of the software architecture to the evolution of software requirements.

Keywords: software evolution, feature modeling, software product lines, software variability, state machines, software architecture, software components.

1. Introduction

This paper proposes an evolutionary software development approach, which uses software product line (SPL) [Clements02, Gomaa05, Pohl05] and feature modeling [Kang90] concepts for evolving multi-version systems. The multi-version systems constitute a family of systems with some common functionality and some variable functionality, a property they share with software product lines. Feature modeling is widely used in software product lines to characterize SPL variability prior to application derivation and deployment. With the approach described in this paper, the goal is to model all versions of the system, including previously deployed systems as a product line. This provides an effective approach for configuration management of multi-version systems.

The emphasis of this paper is on how feature modeling can be used to characterize variability due to software evolution. The approach extends the PLUS UML-based SPL design method [Gomaa05] to address feature-based evolutionary software design. This approach was first suggested in [Gomaa06]. This paper follows up with a more detailed investigation describing how software evolution can be incorporated into a multiple view modeling method using software product line and feature modeling concepts.

2. Evolutionary Software Development

The field of software evolution [MensDemyer08] started with the seminar work of Lehman and Belady [Lehman80] who recognized that software evolution is different from software maintenance. A comprehensive survey of software evolution is given in [Mens08]. Iterative software life cycles such as the Spiral model [Boehm] recognize that software evolution is integral to software development.

The Evolutionary Process Model for SPL engineering used in the PLUS method [Gomaa05] is a highly iterative software process that eliminates the traditional distinction between software development and maintenance. Furthermore, because new software systems are outgrowths of existing ones, the process takes a software product line perspective; it consists of two main processes (see Fig. 1):

a) Evolutionary product line (domain) engineering. A product line multiple-view model, which addresses the multiple views of a software product line, is developed. The product line multiple-view model, product line architecture, and reusable components (referred to as core assets in [Clements02]) are developed and stored in the product line reuse library.

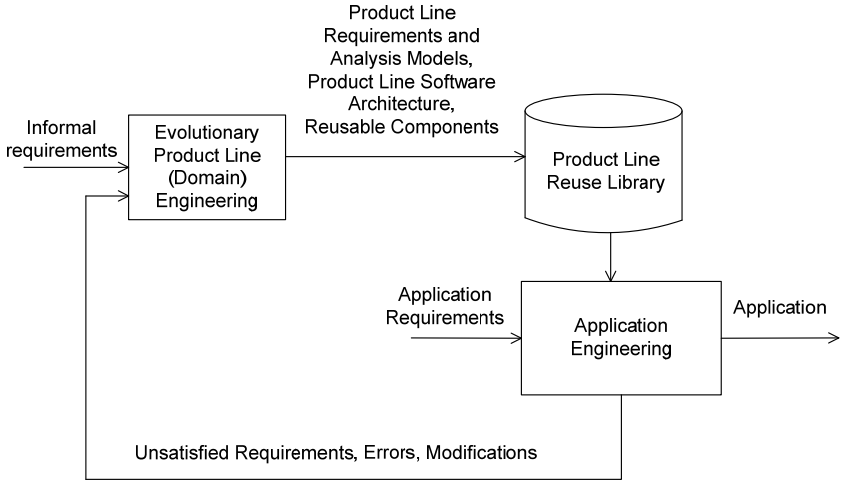


Figure 1 Evolutionary Process Model for Software Product Lines

b) Software Application Engineering. A software application multiple-view model is an individual product line member derived from the software product line multiple-view model. The user selects the required features for the individual product line member. Given the features, the product line model and architecture are adapted and tailored to derive the application architecture. The architecture determines which of the reusable components are needed for configuring the executable application.

The architecture-centric evolution approach described in this paper follows the model driven architecture concept in which UML models of the software architecture are developed prior to implementation. With this approach, the models can later evolve after original deployment. The kernel (base) software architecture represents

the commonality of the product line. Evolution is built into the software development approach because the variability in the software architecture is developed by considering the impact of each variable feature on the software architecture and evolving the architecture to address the feature. The development approach is a feature-driven evolutionary approach, meaning that it addresses both the original development and subsequent post-deployment evolution. Being feature based, the approach closely relates the software architecture evolution to the evolution of software requirements.

The addition of optional and alternative features necessitates the evolution of the original base software architecture by designing optional and variant components to realize these features. This paper describes how feature dependent evolution can be used to systematically incorporate optional and variant components into an evolutionary multi-view model and component-based software architecture.

3. Evolutionary Feature Modeling

Feature modeling is an important concept in software product line engineering [Kang90]. Features are analyzed and categorized as common features (must be supported in all product line members), optional features (only required in some product line members), alternative features (a choice of feature is available) and prerequisite features (dependent upon other features). There may also be dependencies among features, such as mutually exclusive features. The emphasis in feature modeling is capturing the product line variability, since these features differentiate one member of the family from the others [Gomaa05].

Feature modeling can be used to differentiate among the changes to the software system as it evolves through different versions. Thus each version of the system can be described by means of the features it provides. However, during evolution the reuse property of a feature can change. Thus, what constitutes a kernel feature in the first version of the system might evolve into an optional or alternative feature in a subsequent version of the system.

Consider an example of feature evolution. A basic microwave oven system is to evolve by adding some new features, as depicted in Fig. 2 using the UML meta-class notation with stereotypes depicting features and feature groups [Gomaa05]. Using SPL concepts, the original system and evolved system are both members of the SPL. Features provided by both systems are considered common features, while features representing evolving requirements are variable features. The feature variability could be either optional or variant. In the evolutionary microwave oven example, evolution involves adding a Light feature (light is present or not) and a Beeper feature (beep when cooking is finished) to a basic microwave oven. Both features are categorized as optional features from an SPL perspective (Fig. 2), because they only exist in the evolved system. Two other features are added, a multi-line display in place of the original one-line display and a multi-level heating element (high, medium, low) instead of a one-level heating element feature. For these latter two cases, the features are categorized as alternative features (Fig. 2), since in each case a choice has to be made, e.g., between the one-line display and the multi-line display. One of the

alternative features can be chosen as the default feature. From a feature modeling perspective, the initial version of the microwave oven system evolves with the One Line Display feature evolving from a common feature to a default feature in a new Display Unit feature group, in which the alternative Multi Line display feature is added (Fig. 2). The feature group is an exactly-one-of feature group consisting of two mutually exclusive features, one of which must always be selected for a system.

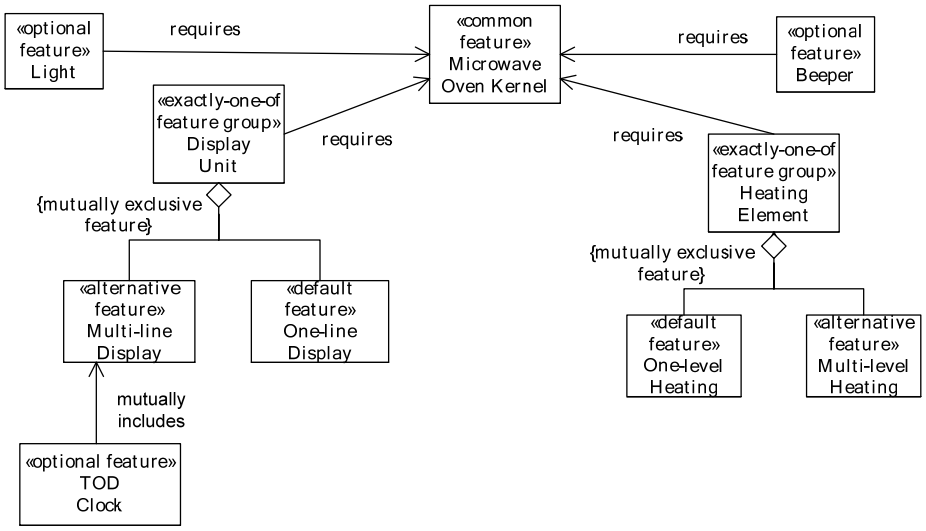


Figure 2 Feature Model for Evolutionary System

4. Evolutionary Use Case Modeling

Whereas feature modeling is effective at differentiating between commonality and variability, use case modeling provides a behavioral perspective on software functionality by describing the sequence of interactions between the actor(s) and the system. It can thus be used to determine where the evolutionary changes need to be introduced in the interaction sequence, which can be taken advantage of during later during evolutional behavioral modeling, as described in Section 5. In particular, during use case modeling, new functionality can be introduced through variation points in the use case description. Variation points can be used to describe newly evolved functionality (represented as an optional feature in the feature model) or evolved alternative functionality (represented as an alternative feature in the feature model). Larger scale evolution could result in new use cases being added, which would be categorized as optional or alternative use cases.

An example from the Microwave Oven case study is the Cook Food use case describing the sequence of interactions between the actor (the user) and the system. The main sequence consists of: the user places an item in the oven, enters the cooking

time, and presses the start button; the system then cooks the food for the specified time. Adding the light feature would result in the variation points introduced for switching on the light when the door is opened and cooking is started, and switching off the light when the door is closed and cooking is finished. Adding the multi-line display would add variation points at steps in the use case where information is displayed on the display. Other evolutionary features such as TOD Clock are sufficiently different that they necessitate the addition of optional use cases, Set TOD Clock and Display TOD, which must be reused together and hence constitute one feature.

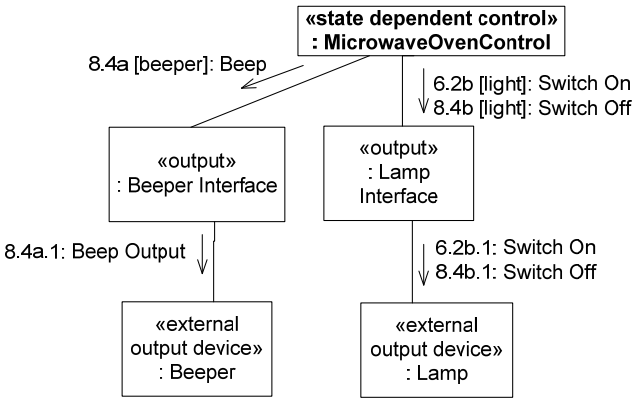
5. Evolutionary Behavioral Modeling

Evolutionary behavioral modeling (also known as dynamic modeling) is an iterative strategy [Gomaa05] to help determine the impact of each newly evolved feature on the software architecture. This results in new components being added to the architecture or existing components having to be modified. The base system is the first version of the evolutionary architecture and corresponds either to the first deployed system or for a product line design, an architecture that consists of the kernel objects.

As the system evolves, it is likely that optional objects (such as Light Interface and Beeper Interface) and variant objects (such as Multi-Line Display Interface) will need to be added. It is also possible that due to evolutionary change, some kernel objects might need to evolve into optional or variant objects.

The evolutionary development approach starts with the base system and considers the impact of optional and/or alternative features. This results in the addition of optional or variant components to the evolving architecture. For example adding the Light and Beeper features necessitates the addition of the optional Light Interface and Beeper Interface. However it also necessitates a change in the Microwave Control object that must send commands to these objects, such as Switch On, Switch Off, and Beep, as shown in the communication diagram in Fig. 3.

Figure 3: Evolution with addition of optional objects



6. Managing Evolution in State Machines

When components evolve, there are two main approaches to consider, specialization or parameterization. Specialization is effective when there are a relatively small number of changes to be made, so that the number of specialized classes is manageable. However, in multi-version system and product line evolution, there can be a large degree of variability. Consider the issue of variability in control classes that are modeling using state machines [Harel96], which can be handled either by using parameterized state machines or specialized state machines. Depending on whether the design uses a centralized or decentralized approach, it is likely that there will be several different state dependent control components, each modeled by its own state machine. The following discussion relates to the evolution within a given state dependent component.

To capture state machine variability due to evolution, it is necessary to specify optional states, events and transitions, and actions. A further decision that needs to be made when using state machines to model variability is whether to use state machine inheritance or parameterization. The problem with using inheritance is that a different state machine is needed to model each alternative or optional feature, or feature combination, which rapidly leads to a combinatorial explosion of inherited state machines. It is often more effective to design a parameterized state machine, in which there are feature-dependent states, events, and transitions. Optional transitions are specified by having an event qualified by a feature condition, which guards entry into the state. Thus Minute Pressed is a feature dependent transition guarded by the feature condition minuteplus in Fig. 4. Similarly, there can be feature-dependent actions, such as Switch On and Switch Off in Fig. 4, which are only enabled if the light feature (Fig. 4) condition is True. Thus the feature condition is True if the optional feature is selected for a given product line member, and false if the feature is not selected. The Beep action is controlled in a similar way. The impact of feature interactions can be modeled very precisely using state machines through the introduction of alternative states or transitions.

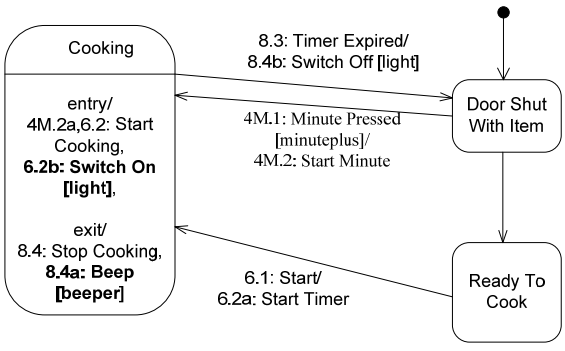


Figure 4 Feature Dependent State Transitions and Actions

Designing parameterized state machines is often more manageable than designing specialized state machines. The feature dependent Microwave Oven state machine provides the overall coordination for the oven since it determine what actions, many of which are feature dependent, are executed and when.

7. Evolutionary Component-Based Software Architecture

A software component's interface is specified separately from its implementation and, unlike a class, the component's required interface is designed explicitly in addition to the provided interface. This is particularly important for architecture-centric evolution, since it is necessary to know the impact of the change to a component on all components that interface to it.

This capability for modeling component-based distributed software architectures is particularly valuable in evolutionary system and product line engineering, to allow the development of kernel, optional and variant components. There are various ways to design components. It is highly desirable, where possible, to design components that are "plug-compatible", so that the required port of one component is compatible with the provided ports of other components to which it needs to connect [Gomaa05]. When plug-compatible components are not practical, an alternative component design approach is component interface inheritance [Gomaa05].

Consider the case in which a producer component needs to be able to connect to different alternative consumer components in different product line members, as shown in Fig. 5. The most desirable approach, if possible, is to design all the consumer components with the same provided interface, so that the producer can be connected to any consumer without changing its required interface. In Fig. 5, the control component Microwave Control (which executes the state machine in Fig. 4) can be connected to either version of the Microwave Display component (which correspond to default and alternative features). The default One-Line Microwave Display and the variant Multi-Line Microwave Display have the same interface, although there is one operation depicted in the IDisplay interface, which is feature dependent and is in fact only realized by the Multi-Line Microwave Display component.

Fig. 6 shows the base component-based software architecture for the Microwave Oven. The base architecture supports the Microwave Oven System use case model with the Cook Food use case before evolution and the introduction of any variation points. The architecture is based on the centralized control pattern [Gomaa11], in which there are input components such as Door and Keypad components, the centralized Microwave Control component, and output components such as the Microwave Display component.

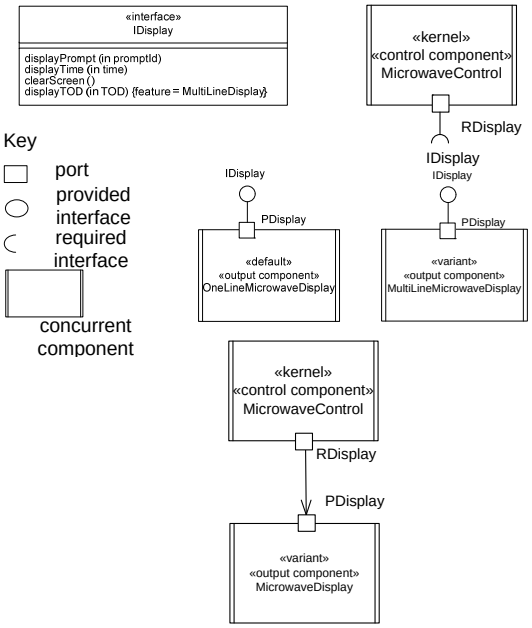


Figure 5 Design of Plug-compatible Components

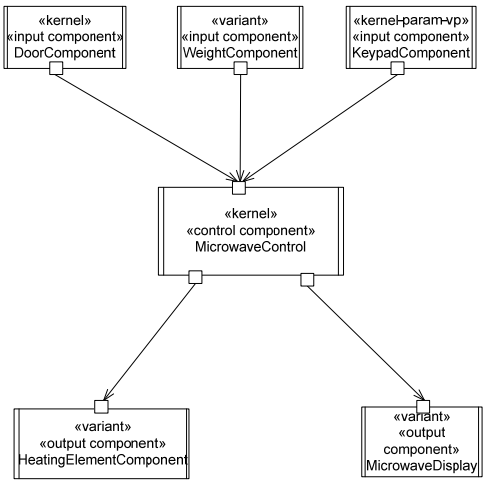


Figure 6 Base Component-Based Software Architecture

Fig. 7 depicts two Feature dependent optional components, Lamp and Beeper, and specifies their interfaces. An evolved software architecture, which includes these two optional components as well as the variant Multi-Line Microwave Display component is shown in Fig. 8.

The control component, Microwave Control, provides the overall coordination for the architecture by executing feature dependent actions given by the Microwave Control state machine (Fig. 4). Microwave Control sends feature dependent messages correspond to these feature dependent actions. Thus, if the Light feature is selected for an application, then the Light feature condition is set to True, resulting in the feature dependent actions Switch On and Switch Off being enabled when the transitions into and out of Cooking state take place. Microwave Control will send corresponding feature dependent messages to the Light component when these actions occur, which will invoke the Switch On and Switch Off operations respectively in the Light component Fig. 7).

8. Conclusions

This paper has described an evolutionary software development approach, which uses software product line and feature modeling concepts for evolving multi-version systems. The multi-version systems constitute a family of systems with some common functionality and some variable functionality. With the approach described in this paper, the goal is to model all versions of the system, including previously deployed systems as a software product line. The addition of optional and alternative features necessitates the adaptation of the original software architecture by designing optional and variant components to realize these features**.**

Future work involves exploring further how to incorporate the notion of history, i.e., the sequence of evolution. This could be addressed by assigning each application a version number (suitable for linear evolution) and/or providing links between successive versions (suitable for branching evolution) of the evolving system. To ensure consistency among the multiple views of an evolving system, it is desirable to have an underlying multiple view meta-model. A multiple view meta-model for product lines has been developed [Gomaa08]. Future work involves extending this meta-model to incorporate concepts of software evolution.

This paper has described evolutionary design of component-based software architectures at design time. It is also possible to extend this research to address dynamic run-time adaptation [Gomaa07] of the evolved system. Although the paper has described feature-based evolution of component-based software architectures, the concepts can also be applied to service-oriented architectures [AbuMatarGomaa11].

Figure 7 Feature dependent Optional Components

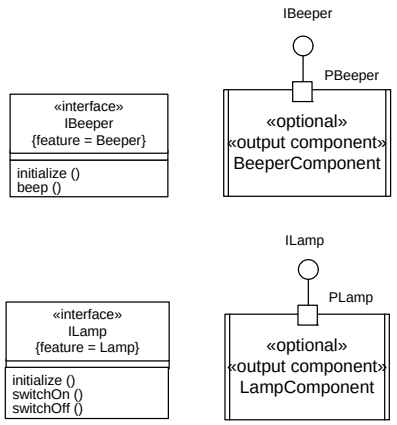
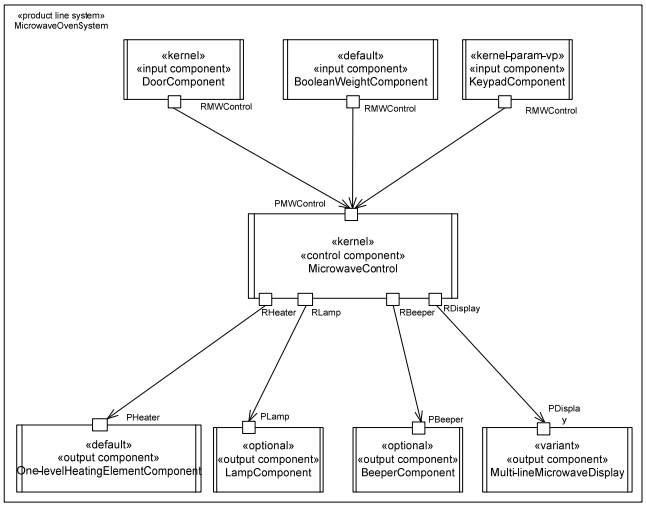


Figure 8 Evolutionary Component-Based Software Architecture



References

[Abu-Matar10] M. Abu-Matar and H. Gomaa, "Feature Based Variability for Service Oriented Architectures", Proc. Wkshp. on Variability in Software Architecture, Boulder, CO, June 2011.

[Boehm88] Boehm, B., "A Spiral Model of Software Development and Enhancement," IEEE Computer, May 1988.

[Clements02] P. Clements and L. Northrop, Software Product Lines: Practices and Patterns, Addison Wesley, 2002.

[Gomaa05] Gomaa, H., "Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures", Addison-Wesley, 2005.

[Gomaa06] H. Gomaa, "A Software Modeling Odyssey: Designing Evolutionary Architecture-centric Real-Time Systems and Product Lines", Proc. 9th International Conf. on Model-Driven Engineering, Languages, and Systems, Genova, Italy, October 2006.

[Gomaa07] H. Gomaa and M. Hussein, "Model-Based Software Design and Adaptation", Proc Intl. Wkshp. on Sftwr. Eng. for Adaptive and Self-Managing Systems, Minneapolis, May 2007**.**

[Gomaa08] H. Gomaa and M.E. Shin, "Multiple-View Modeling and Meta-Modeling of Software Product Lines", Journal of IET Software, Vol. 2, Issue 2, pp. 94-122, April 2008.

[Gomaa11] H. Gomaa, "Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures", Cambridge University Press, February 2011.

[Harel96] Harel, D. and E. Gary, "Executable Object Modeling with Statecharts", Proc. 18th International Conference on Software Engineering, Berlin, March 1996.

[Kang90] Kang K. C. et. al., "Feature-Oriented Domain Analysis," Technical Report No. CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[Lehman80] M.M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution", Proc. IEEE 68 (9): 1060–1076, 1980.

[MensDemeyer08] T. Mens, S. Demeyer (Eds.) Software Evolution. Springer, 2008.

[Mens08] T. Mens, Introduction and Roadmap: History and Challenges of Software Evolution. In Software Evolution, Springer, 2008.

[Pohl05] K. Pohl et al, "Software Product Line Engineering: Foundations, Principles and Techniques", Springer, 2005.