Preliminary Proceedings

International Workshop on Models and Evolution
ME 2010
ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems
Oslo, Norway - October 3-8, 2010
http://www.modse.fr

Program Chairs
Dirk Deridder
Vrije Universiteit Brussel, Belgium
Bernhard Schätz
fortiss GmbH, Germany
Dalila Tamzalit
LINA, University of Nantes, France
Alfonso Pierantonio
Università degli Studi dell’Aquila, Italy

Steering Committee
Dirk Deridder
Vrije Universiteit Brussel, Belgium
Hubert Dubois
CEA LIST, France
Jeff Gray
University of Alabama, USA
Tom Mens
Université de Mons, Belgium
Alfonso Pierantonio
Università degli Studi dell’Aquila, Italy
Bernhard Schätz
fortiss GmbH, Germany
Pierre-Yves Schobbens
University of Namur, Belgium
Dalila Tamzalit
LINA, University of Nantes, France
Stefan Wagner
TU München, Germany

Program Committee
Arnaud Albinet
Continental Automotive, France
Benoît Baudry
IRISA, University of Rennes, France
Mireille Blay-Fornarino
Université de Nice-Sophia Antipolis, France
Jean-Michel Bruel
University of Toulouse, France
Jordi Cabot
EMN Nantes, France
Rubby Casallas
University of Los Andes, Colombia
Antonio Cicchetti
Mälardalen University, Sweden
Davide Di Ruscio
Università degli Studi dell’Aquila, Italy
Anne Etien
INRIA Futurs, France
Jean-Marie Favre
OneTree Technologies, Luxembourg
Mehdi Jazayri
University of Lugano, Switzerland
Gerti Kappel
TU Wien, Austria
Udo Kelter
Universität Siegen, Germany
Olivier Le Goaer
LINA, University of Nantes, France
Richard Paige
University of York, United Kingdom
Antonio Vallecillo
Universidad de Malaga, Spain
Ragnhild Van Der Straeten
Vrije Universiteit, Belgium
Hans Vangheluwe
Universiteit Antwerpen, Belgium
Dennis Wagelaar
Vrije Universiteit Brussel, Belgium
Merging Model Refactorings
- An Empirical Study -

Maximilian Koegel, Markus Herrmannsdoerfer, Otto von Wesendonk,
Jonas Helming, and Bernd Bruegge

Institut für Informatik, Technische Universität München
Boltzmannstrasse 3, 85748 Garching, Germany
{koegel, herrmama, wesendon, helming, bruegge}@in.tum.de

Abstract. Model-driven development promotes models as abstractions of a system from which the system code can be generated. As a consequence, many development activities such as refactoring have moved from the code- to the model-level. A model refactoring is a composite change—consisting of a series of model changes—that improves the model quality but does not change the system behavior. Version Control (VC) systems are applied to control change on models and to facilitate collaboration. Many of them are limited in their ability to detect composite changes and to use them for merging. To overcome these limitations, operation-based VC systems directly track the composite changes. In this paper, we present an empirical study to answer the following research questions: (1) Do developers produce a better merge result when merging with information about composite changes? (2) Do developers favor to discuss complex merges instead of being forced to solve them immediately? We show that merging can be improved significantly by information about composite changes, and that developers like to discuss merges when they reject changes from the repository.

1 Introduction

Models are an essential artifact throughout the entire life cycle in software engineering projects. These models include UML and SysML models as well as project management models like release plans. Model-driven software development is putting even more emphasis on models, since they are not only an abstraction of the system under development, but the system is generated from its models. As a result, many development activities that have originally been performed on code are now performed on models. One of these activities is refactoring. Model refactorings are model transformations that restructure the models in order to improve certain quality attributes of the models, but preserve the behavior [19]. They are the model-level equivalent of source code refactorings [7]. Refactoring is a maintenance activity. According to Lehman [13], an evolving system erodes due to increasing complexity without refactoring. Model refactorings are complex changes on a model and are usually comprised of several atomic model changes. For example, an extract superclass refactoring on
a UML class diagram creates a new superclass for a number of existing classes and moves shared attributes and associations from the subclasses to the new superclass. This refactoring results in many atomic changes.

Version Control (VC) systems—also known as Software Configuration Management systems—are widely used to control change on models and to facilitate collaboration. Most of them employ optimistic concurrency control to allow for parallel and offline work on their configuration items which may be models. As a consequence, conflicts can occur between the changes that are performed concurrently by different parties on the configuration items. Model merging is the activity of resolving these conflicts. Traditional VC systems such as CVS or Subversion support merging for textual artifacts such as source code on a textual level. However, many researchers recognized that these systems are very limited when applied on models, especially in the use case of merging [1, 5, 10, 11, 16–18]. Nguyen et al. describe this problem as the impedance mismatch between the flat textual data models of traditional VC systems and graph-based software models [16]. Different approaches have been proposed to cope with the shortcomings of existing methods and techniques to better support change tracking and merging of graph-based models. They can be categorized into three different classes: state-based, change-based and operation-based approaches [4].

- **State-based approaches** only store states of a model and thus need to derive differences by comparing two states—e.g., a version and its successor—after the changes occurred. This activity is often referred to as differencing and is required for merging to detect conflicting changes.
- **Change-based approaches** record the changes, while they occur, and store them in a repository. There is no need for differencing, since the changes are recorded and stored, and thus do not need to be derived later on.
- **Operation-based approaches** are a special class of change-based approaches which represent the changes as transformation operations on a state. The recorded operations can be applied to a state to transform it into the successor state. Merging is performed based on the recorded operations.

From a versioning point of view, refactorings are complex changes and are more difficult to cope with than simple changes, such as changing an attribute or creating an element. Refactorings should be considered as atomic changes for versioning, since they should not be applied partially. However, most of the state-based approaches are not able to detect refactorings, since they have to derive the changes by differencing. It is impossible to detect a refactoring with certainty, since it looks exactly like a set of simple changes. Nevertheless, there are approaches that try to heuristically recover the refactorings from a state-based difference [2, 3, 8]. In contrast, operation-based VC systems can directly record refactorings as composite operations.

Independent of the approach how the information about composite changes was derived, it is considered to be helpful for conflict detection and merging [14, 15]. However, to our best knowledge, there is no empirical evidence about the benefits of information about composite changes for model merging. To provide
this evidence, we have performed an empirical study. We have developed a question-naire consisting of 8 merging scenarios and distributed it to 46 developers. As a result, we received 778 merge decisions and statistically analyzed them. In this paper, we present the design and results of the empirical study.

Outline. Section 2 provides an introduction to composite changes on models and to the merge problem considering these changes. Section 3 presents the design of the empirical study, its results and an interpretation of the results. Section 4 concludes the paper with a short summary. Related work is not aggregated in a separate section but is provided inline.

2 Merging Complex Changes

Models are abstractions of the system under development or of the development project itself. Models are essentially graphs with attributed nodes. Many of the possible changes on a model can be described by simple graph changes, such as adding edges and nodes or changing attributes of nodes. However, some models are also subject to more complex changes. For example, a UML class diagram can be refactored. Within one command from a user—e.g., performing an extract superclass refactoring—many nodes and edges might be created, removed or changed. From a technical point of view, these changes consist of a number of simple changes. This implies that changes occur on different levels of abstraction; we refer to these levels as change granularity.

While traditional VC systems treat change on the representational level, approaches exist on top of them to treat changes on the level of a graph representing the model [6]. If the respective approach is not independent of a certain metamodel, it may also treat changes on the metamodel level, e.g., on the level of a UML class diagram. To capture refactorings on models the system must also be able to capture complex changes as composite changes consisting of several changes on the metamodel level of change. Furthermore, many of these complex changes are performed atomically. So they only reflect the developer’s intention, if all the included changes on the model are applied as an atomic unit. Otherwise, the intention of the developer—such as preserving the behavior in case of refactoring—is only partially reflected in the model and hence might not make sense. To illustrate a conflict on a complex change and a merge result with a partially applied complex change, we provide a detailed example from the domain of UML class diagrams. The complex change is an extract superclass refactoring. This refactoring creates a common super class for a number of existing classes. All common attributes and associations of the existing classes are pulled up into the newly created superclass. The ancestor model as well as all other versions are shown in Figure 1.

We assume that a developer A now performs an extract superclass refactoring on the classes Bus and Car, resulting in the model state shown in Figure 1. The refactoring effectively pulls up the composition to Engine from Car and Bus to the new superclass PoweredVehicle. Another developer B also starts
with the same ancestor model but decides to perform a different change, resulting in the model state shown in Figure 1. The developer changes the multiplicity of the composition from Bus to Engine on the side of Engine to unbounded to reflect the fact that buses might have more than one Engine. Under the assumption that developer A has already committed his changes to the repository, an attempt to commit from the second developer B will result in a merge situation. The two design choices and hence the two intentions of the developers are conflicting. Either the composition from Car to Engine has a different multiplicity than the composition from Bus to Engine, or they have the same and the refactoring can be fully applied. If a system is not aware of intentional changes, it will look at this conflict from a metamodel level point of view. Developer A wants to delete the composition from Bus to Engine (as part of the refactoring), while developer B wants to change the multiplicity of the same association on one end. This can also be identified as a conflict – however, it is not only displayed differently to the user, but it is also treated differently in a merge. In case developer B decides to keep his or her change, the refactoring of developer A will only be partly reflected. The composition from Bus to Engine is still required to exist to be able to keep the change of developer B. However, Bus now additionally shares another composition with Engine inherited from its new superclass PoweredVehicle. The resulting model with the partially reflected refactoring is shown in Figure 1. In the following, we refer to these partially applied refactorings as partial refactorings. Partial refactoring shall be avoided, because refactorings represent an atomic set of changes and shall not be applied partially.

Fig. 1. Example of a partially applied Refactoring after the Merge

3 Empirical Study

In this section, we present the empirical study to show the benefits of information about composite changes for model merging. We describe the research questions, the design of the study, the results, their statistical evaluation and interpretation.
3.1 Research Questions

We conducted the empirical study to answer the following research questions:

1. **Do developers produce a better merge result when merging with information about composite changes?**
   When developers merge models with complex changes such as refactorings, do they profit from information about composite changes? And if they do, how much do they profit? Is the merge result better in comparison to a merge result without this additional knowledge?

2. **Would developers favor to discuss complex merges instead of being forced to solve them immediately?**
   When developers merge models, they are normally forced to complete the merge, before they can commit their changes to the repository. Would developers favor to postpone a merge decision and first discuss the conflict with the involved developer?

3.2 Study Design

We designed a questionnaire consisting of 8 different merging scenarios. In every scenario, the developer is presented a description of the current model state and of the changes that he or she has performed including a rationale for the changes. The developer is then confronted with a merge situation due to conflicts with new incoming changes from the repository.

The model is a simple UML class model in slight variations, see Figure 1 for an example. The conflicting changes always contain at least one refactoring operation as a representative for a complex change. We selected the refactorings by frequency of occurrence in a case study on the evolution of the metamodels from the Graphical Modeling Framework (GMF) that are specified in Ecore [9]: Extract/Inline Class, Extract/Inline Superclass, Partition/Join Association and Pull-up/Push-down Attribute/Association. Figure 2 shows an example for each refactoring.

![Fig. 2. Refactoring Examples](image-url)
The same scenario is once presented without the composite changes and once with the composite changes taken into consideration. In other words, the merge is once performed on the intentional level of change, and once on the metamodel level of change. Since the time for questions per interviewee is limited, we can not present models and also commits with realistic sizes. To simulate reasonably large commit sizes, we just presented the conflicting changes, but gave the developers the choice to look at the other, non conflicting changes which might help to understand the intention of the other developer. Therefore, the developers had the following choices: (1) **Accept Mine**: Accept my change and reject the change from the repository, (2) **Accept Theirs**: Accept the change from the repository and reject my change, and (3) **Take More Time**: Take more time to look at all incoming changes.

For each merge result, we measure whether it implies a partial refactoring as introduced in Section 2. Please note that for questions that take composite changes into account, it is impossible to perform a partial refactoring. If a conflicting change as part of a composite change is rejected, the whole composite will be rejected. Additionally, we offer the developer for every merge question the possibility to discuss his or her choice with the respective opponent developer. The merge scenarios and questions were not represented in a tool to avoid any representational or usability-related influences on the result. The questionnaire is available online at [12].

### 3.3 Study Result

We have interviewed 46 developers by means of an electronic questionnaire. 48% of the developers are professionals and 52% students. 67% of the professionals are software engineers, 29% researchers and 4% other developers. 58% of the students are Bachelor students and 42% Master students, and they all have a Computer Science related major. Figure 3 depicts the experience of developers in years grouped by three areas: Object-Oriented Programming, Source Code Repositories and Unified Modeling Language and the frequency of use of refactorings on code and on UML class diagrams.

![Developer Experience and Refactoring Usage](image)

**Fig. 3.** Developer Experience and Refactoring Usage

The 46 developers have answered 778 merge questions in total. 368 of those were questions with information about composite changes and 410 without. For
the latter, 49% of the merges resulted in partial refactorings. Conservatively, we assumed that a developer selecting to take more time to look at all incoming changes would come up with a merge without partial refactorings. As noted earlier, there are no partial refactorings with the information about composite changes, which was also reflected in the data. Figure 4 shows the distribution of the choices for the merge scenarios for questions with and without information about composite changes. The developers also indicated whether they would like to discuss a decision with the developer that committed the conflicting change. The full results of the study in terms of raw data are available on [12].

![Fig. 4. Merging and Discussion Choices](image)

### 3.4 Statistical Evaluation

We have performed a number of statistical tests on the data to test the validity of different hypotheses in terms of the gathered data. All the tests we performed are on binary random variables. We assume that the decisions for different merge questions are independent. Therefore, the binary random variables can be assumed to have a binomial distribution. Binomial distributions are characterized by a probability \( p \) for a single experiment to succeed. For all tests we have two sample groups. For example, we have the groups of merge decisions with and without information about composite changes, and the binary variable whether the merge produced a partial refactoring or not. For the two samples, we calculate a 99% or 95% confidence interval for the binomial distribution’s probability \( p \). In other words, given the sample the parameter \( p \) of the distribution is with 99% or 95% probability in this interval. We calculate the interval based on the following formula by using the arithmetical average as a maximum likelihood estimate for \( p \), where \( k \) is the number of instances of the sample where the variable is true and \( a \) is the constant for the respective significance level (1.96 for 95% and 2.3264 for 99%):

\[
\left| \frac{k}{n} - p \right| \leq a \times \sqrt{\frac{p \times (1 - p)}{n}}
\]

Once we have derived the interval for both samples, we check if they overlap. If they do overlap, we cannot derive any significant statement for the hypothesis. If they do not overlap, however, we can accept the hypothesis on a 99% or 95%
significance level. In the following, we will describe different hypothesis and their results:

**Partial Refactorings vs. Composite Changes.** Our hypothesis is that partial refactorings are significantly less frequent in merges with information about composite changes. We have two samples: all merge decisions without and with information about composite changes. The binary random variable is true if there is no partial refactoring. The 99% confidence interval for the group without composites is $[0.4506, 0.5708]$, while the other group results in the interval $[1, 1]$. Clearly, they are not overlapping, so we can accept the hypothesis on a 99% significance level.

**Merge Choice vs. Composite Changes.** Our hypothesis is that the distribution of the merge choices ($Accept\ My$, $Accept\ Theirs$, $Take\ More\ Time$) was different for merge decisions without and with information about composite changes. More specifically, we expected more choices to $Accept\ Mine$ for decisions without information about composite changes. However, the intervals are overlapping, so we can not accept the hypothesis on a 95% significance level.

**Discussion vs. Merge Choice.** We expected that developers would discuss decisions more often, if they reject a change from the repository or if they request more time. This leads to three hypotheses on this correlation. Our first hypothesis is that merge decisions with the choice $Take\ More\ Time$ required a discussion more often than decisions with the choice $Accept\ Theirs$. Our second hypothesis is that merge decisions with the choice $Accept\ Mine$ required a discussion more often than decisions with the choice $Accept\ Theirs$. Finally, our third hypothesis is that merge decisions with the choice $Accept\ Theirs$ required a discussion less often than both other decisions. We have two samples for each hypothesis: all merge decisions with the respective choice and all decisions with another choice. The binary random variable is true if there is a discussion request for the decision. The 99% confidence intervals for the first groups are $[0.6080, 0.7369]$, $[0.6157, 0.7426]$ and $[0.1517, 0.2350]$, respectively. The intervals for the second groups are $[0.1199, 0.2351]$, $[0.2091, 0.3353]$ and $[0.5653, 0.6927]$. The two corresponding intervals for all three hypotheses do not overlap, and we can thus accept the three hypothesis on a 99% significance level.

### 3.5 Result Interpretation

In this section, we interpret the results in terms of the research questions posed in Section 3.1, taking into consideration the statistical results from the previous section.

The results show that developers produce a fairly large percentage (about 50%) of partial refactorings in their merge results, if they do not have information about composite changes. They decided in favor of them, even though they had the choice of taking more time to look at all incoming changes. We claim that this is a realistic simulation of the merge case, where possibly hundreds or thousands of changes are incoming that might help to understand the conflicts. In these situations, developers are forced to either spend time to look at many changes or to make a decision only by looking at the conflicting changes. Also,
we claim that partial refactorings are a suboptimal merge result, and that it is favorable to apply a refactoring completely to preserve its intention. The statistical test provides evidence that partial refactorings are less frequent in merges with information about composite changes on a significance level of 99%. More specifically, the difference in the probability of partial refactorings is at least 0.43 with a significance of 99%.

In 70% of these complex merge situations, developers would have liked to discuss their choice. The request for discussion is dependent on the choice the developers made in the merge. If they choose to accept their changes or take more time for a decision, they tend to discuss the choice more often. A selection to accept the incoming change is less often selected for discussion. Both claims have been shown on a 99% significance level by the statistical tests. This seems reasonable and we expected this, since a developer only favors an incoming change over his or her change, if he or she understands and approves the incoming change. These decisions less often require a discussion. If a developer selects his or her own change and thus discards another change, it seems reasonable to discuss this choice with the other developer more often. Finally, if a developer chooses to take more time, the decision seems to be difficult and therefore is probably more often suited for a discussion.

4 Conclusion

We investigated the problem of merging complex changes with and without information about composite changes. We assessed both approaches in an empirical study. During the study, 46 professional and student developers made 778 decisions on merges including refactorings. The results clearly show that information about composite changes significantly improves the merge result. We can conclude on a 99% significance level that the probability for a consistent merge result on complex changes is at least 75% better with information about composite changes than without it. Also we can show that developers favor to discuss complex merges when they reject changes from the repository. A solution based on operation-based change tracking to capture change on the intentional level was already presented in [10, 11]. The results of this study imply that applying operation-based merging can improve merge results.

References

Documenting Stepwise Model Refinement using Executable Design Decisions

Matthias Biehl
Embedded Control Systems
Royal Institute of Technology
Stockholm, Sweden
biehl@md.kth.se

Abstract. During model refinement a wealth of knowledge about the model under development is accumulated that is only partly represented by the model itself. Design decisions and the considered modeling alternatives are neither represented by the model nor are they documented. During later lifecycle stages this information is often not available any more, which reduces the understandability of the model and potentially leads to inconsistencies and erosion of the model. We propose an approach to capture and store the design decisions in model-driven development. We represent design decisions as model transformations and propose tool support that applies this representation to capture design decision with low effort. The captured design decisions provide a record of the model evolution and the rationale of the evolution.

Key words: Design Decision, Evolution, Model Transformation, Refinement

1 Introduction

During the early stages of architectural design a number of alternative architectural models are created, evaluated and refined. During this exploration phase a lot of knowledge about the system is acquired, which is important during later lifecycle phases such as detailed design and maintenance. However, this knowledge consisting of design decisions and their justification is usually not documented due to the associated time and cost overhead.

Capturing and documenting design decisions is expensive and is often seen as an additional obligation not providing any immediate value in the design process. In fact, the value of design decision documentation will usually surface much later in the lifecycle, for example during maintenance activities. By the time the knowledge is needed, it is usually not readily available any longer. Decisions that seemed to be self-evident at the time they were made can often not be reconstructed or understood later in the development process. This problem is known as knowledge vaporization [3], which results in further problems such as inconsistencies, design erosion, aging and architectural drift [20, 16, 17].
The design method of stepwise refinement of programs was first explored by Dijkstra [6] and by Wirth [22]. Wirth states that program development is a series of refinement steps, further he links refinements with the design decisions. Refinement can have slightly different meanings. In the strict sense, a refinement describes adding detail to something that already exists. In a more relaxed sense, refinement can also include adding new functionality. An example of a development method based on architectural refinement is Attribute Driven Design (ADD) [23]. ADD prescribes a top-down decomposition process that subsequently attacks all quality attributes in their order of importance.

In this work we assume a model-driven development process, where models are developed iteratively by refining a baseline model. Iterative refinement starts with an initial model, which is the first baseline model and may be empty. In each refinement step the baseline model is changed, yielding a new model that serves as baseline for the consecutive refinement step.

One refinement step entails a small change in the model and cannot be understood without considering related refinement steps made immediately before and after. It makes sense to bundle refinement steps into logical entities that comprise several refinement steps. A design decision is such a logical entity that comprises several related refinement steps.

When a model is refined, a number of design decisions are made and applied on an initial model. As a result a modified model incorporating the changes induced by the design decisions is created. Thus the model refinement can be documented by a sequence of design decisions.

Architects have long recognized the importance of documenting design decisions, as Tang et al. found in their survey on architecture design rationale [18]. Practitioners use documented design rationale to reason about the architecture. While some representations and tool support are available, the capturing of design decisions and their rationale still creates additional overhead, making design decision documentation a double edged sword [10].

This paper is structured as follows. In section 2 we introduce related work. In section 3 we introduce our representation for design decisions. In section 4 we show how this representation can be used to capture design decisions with low effort. In section 5 we present a case study in which we apply our approach on a model of an automotive embedded architecture. We conclude this work in section 6.

2 Related Work

In this section we review related work in the area of design decision representation. Kruchten [9] specifies an ontology of architectural design decisions. A number of different approaches for representing design decisions have been proposed in academia. Informal representations document design decisions in the form of free text and use-cases. Template-based approaches provide a guideline for textual descriptions of design decisions such as the template by Tyree and Akermann [21]. Structural approaches explicitly represent the design delibera-
tion process, including alternatives and arguments. Examples are IBIS [11], PHI [15], DRL [12] and QOC [14]. Approaches for representing design decisions that were developed in recent years focus on linking design decisions with the architectural model. The SEURAT approach links rationale to specific lines in the source code [4]. AREL is a UML profile that can be used to annotate architectural models with design decisions and rationale [19]. The Archium [7] approach associates design decisions with modifications of the architecture. These models provide descriptions, traceability and linking to other modeling elements.

3 Representation of Design Decisions

In this section we introduce our representation for design decisions, which is tailored for model-driven development. There are several objectives for our representation of design decisions for model-driven development:

- **Objective 1**: Design decisions should be represented explicitly.
- **Objective 2**: The design decision representation should reduce the capture overhead.
- **Objective 3**: Design decisions should be represented independently of a certain modeling language.
- **Objective 4**: The documentation of design decisions should be consistent with the architectural model.
- **Objective 5**: The documentation of design decisions should be linked to the model elements affected by the decision.

3.1 Executable Design Decision

Our representation for a design decision consists of the following parts: a model transformation and a textual description (objective 1). The representation is called *executable design decision* [1], as a model transformation engine can execute it automatically (objective 2). The core of the executable design decision is a model transformation, which describes both the context in which the design decision can be made and the effect that the design decision will have on the architecture (objective 5). Since we use model transformations to represent design decisions, we can execute the design decision and inspect the effect the design decision has on the model. As a consequence of this automation, the model and the design decision are consistent (objective 4). Model transformations can be specified for any metamodel without needing to change the metamodel (objective 3). Different model transformation technologies can be used for expressing design decisions, as long as the model transformation engine fulfills certain requirements, such as support for model-to-model transformations that are endogenous and in-place.
3.2 Formalization of Design Decisions

In the previous section we introduced our representation of design decisions. Design decisions are interpreted as change functions and represented as model transformations. When a design decision is made for a given model, the result is a new model incorporating the changes induced by the design decision. If we interpret the design decision as a function, the input of the function is the previous model, the output is a new model.

This can be formalized as follows: A design decision \( d : M \rightarrow M \) made on a given architectural model \( A_0 \in M \) yields a new architectural model \( A_1 \in M \), where \( M \) is the set of possible architectures described according to a common metamodel. The design decision can be interpreted as a function over architectural models, where an execution of the design decision creates a new architecture incorporating the changes induced by the design decision \( d \), in short \( A_1 = d(A_0) \).

Several design decisions \( d_1 \ldots d_n \) are made during architectural refinement, formalized as a composition of functions \( A_n = (d_n \circ d_{n-1} \circ \ldots \circ d_1)(A_0) \). As a result we can describe the design process explicitly as \( d_n \circ d_{n-1} \circ \ldots \circ d_1 \), i.e. the sequence of steps by which the architecture is refined, instead of the outcome \( A_n \) of the design process.

4 Capturing Design Decisions

In stepwise model refinement the capturing of design decisions comprises four steps:

1. Performing an update of the model, so it reflects the new design decision
2. Documenting where in the model the change applies and what is changed
3. Documenting the rationale of the change
4. Linking the documentation to the model elements affected by the change

The overhead of these steps is one of the main reasons why design decisions are not documented [10]. Tools can help to reduce the overhead for the architect, by exploiting the intrinsic relation between steps (1), (2) and (4). A model transformation has a precondition and a postcondition that can be used for documenting where in the model the change applies and what is changed (2). When the model transformation is executed on the initial model, the outcome is the updated model (1). The precondition of the model transformation links the design decision to the model before the change and the postcondition links the design decision to the model after the change (4). A model transformation is capable of capturing steps (1), (2) and (4) by a single representation.

The architect does not need to create the model transformation manually, instead it can be created algorithmically from different snapshots of the model. We can create this model transformation based on a snapshot of the model taken before the change and a snapshot taken after the change. We use a triple graph grammar (TGG) rule to describe the design decision \( d_n \) of section 3.2. A TGG rule consists of three graphs, a left-hand side graph, a right-hand side graph and a mapping graph. We calculate these three components in the following way:
- **Left-hand Side (LHS)** The model $A_{n-1}$ corresponds to the context of the applied decision, i.e. the snapshot of the model before the change.
- **Right-hand Side (RHS)** The model $A_n$ corresponds to the outcome of the applied decision, i.e. the snapshot of the model after the change.
- **Mapping** The mapping graph establishes identity links between elements in the LHS and RHS. Given the LHS and RHS, the mapping graph can be computed e.g. by comparing the values of the model elements in LHS with the model elements in RHS.

5 Case Study

To validate our proposal of using model transformations to represent design decisions we perform a case study on design decisions in a model of an automotive brake-by-wire system. In this case study we explore modeling a design decision using different model transformation languages, **ATL** and **Tiger EMF**. ATL is a representative from the class of hybrid (declarative and imperative) model transformation languages and Tiger EMF as a representative from the class of model transformations based on triple graph grammars. The intention is to explore the feasibility and usability of different classes of model transformation approaches for representing design decisions.

5.1 Brake-By-Wire Architecture

In modern cars the traditional mechanics and hydraulic connection between user interface and actuators are increasingly replaced by electronics and software [13]. In this case study we will look at a subsystem of the automotive software architecture, a brake-by-wire system.

We use EAST-ADL2 [5] as a representation for the architecture of the brake-by-wire system. EAST-ADL2 is an architecture description language for the domain of automotive embedded systems. It can be used to describe high-level abstract functions and both the hardware and the software architecture of an embedded system.

Figure 1 shows an initial architecture of the brake-by-wire system modeled with EAST-ADL2. We use it as an initial model that is subsequently refined. In a brake-by-wire system the brake pedal is a component that reads the user’s input and sends this information over a network to the brake controllers at every wheel. At each wheel a brake controller receives both information about the wheel speed and the requested braking from the pedal. The brake controller actuates the mechanical brake in order to achieve a good braking performance, e.g. to avoid skidding.

Safety and reliability are important qualities of a brake-by-wire system. Thus the design decisions made during the development of the brake-by-wire system need to address safety. Common design decisions in safety-critical embedded systems are decisions about the replication of components with the intent to
enhance the reliability of the system. These decisions have a major impact on both the safety and the cost of the complete system as we will see in this example.

Option B in Fig. 2 shows a refined architecture of the brake-by-wire system. Here we intended to increase the reliability and safety of the system and decided to use triple redundancy with a voter for the pedal sensor. An alternative design option uses dual redundancy instead, depicted as option A in Fig. 2. The second sensor serves as a standby, running in parallel with the primary sensor, so that at least one of the components delivers a signal, should the other one fail. Dual redundancy improves reliability compared to the initial model and does not have the high cost of triple redundancy.

5.2 Representation of Design Decisions with Tiger EMF

Tiger EMF [2] is a model transformation framework that is based on triple graph grammars. It supports in-place transformations. A transformation rule is specified using a pattern to be matched in the source model (left-hand side), and a pattern to be created in the target model (right-hand side). Between elements from the left-hand side and elements from the right-hand side a mapping-relation, the third graph, can be specified, which is expressed with corresponding numbers in the left-hand side and right-hand side.

The context of the design decision corresponds to the left-hand side of the model transformation, depicted in Fig. 3.
The outcome of the design decision corresponds to the right-hand side, depicted in Fig. 4. The negative application condition specifies that the transformation is applied at most once, and due to space constraints it is not included here. The transformation creates an architecture with triple redundancy as depicted in Fig. 2 B.

We can use existing tools for reliability and cost analysis on this architecture. Even though the reliability of the system is improved, the component costs for triple redundancy with voter are high. We might want to explore an alternative design decision for double redundancy. Based on the initial architecture (see Fig. 1) and the alternative design decision, we can generate an alternative architecture.

We create an alternative design decision using a new model transformation. The right-hand side of the new model transformation is depicted in Fig. 5, the left-hand side is the same as for the previous model transformation, as the decision is applied in the same context. After executing the model transformation, an architecture with double redundancy is created (see Fig. 2 A).
As we have seen in this example, the design space can be explored without much effort by reusing the design decisions and modifying them slightly. Design decisions solving the same issue differently, i.e. alternative design decisions (or model transformations), have the same context (or left-hand side) and a different outcome.

Moreover, we can change the design decisions to be applicable in a different context. The design decision for double redundancy (Fig. 5) can be applied to the WheelSpeedSensor instead, by changing the string PedalSensor to WheelSpeedSensor in the corresponding left-hand side (Fig. 3). This can be generalized by introducing parameters into the transformation.

5.3 Representation of Design Decisions with ATL

To show that design decision can be represented with different model transformation languages, we created a second example of the design decision for triple redundancy and implemented it in ATL. The ATLAS Transformation Language (ATL) [8] is a hybrid model transformation language, supporting both declarative and imperative programming styles. An ATL transformation is composed of rules that describe how to create and initialize the elements of the target models. To support in-place transformations, ATL has to be configured to run in a special execution mode, the refining mode. Several practical limitation of the current ATL refining mode exist, such as missing foreach-loops and lazy rules. Due to space constraints we have not included the transformation source code.

6 Conclusion

We proposed model transformation as a representation for design decisions fulfilling the identified objectives for design decisions in model driven development. We call this representation executable design decisions, since an engine can automatically create the changed model based on this design decision representation.
Design decisions can be captured based on two snapshots of the model: (1) the model before the change and (2) the model after the change. The model transformation representing the design decision can be computed. The architect does not have to code the transformation manually.

In a case study we have shown that it is feasible to represent executable design decisions using different model transformation languages. We noticed differences between the languages. Creating the model transformation graphically, such as in Tiger EMF, resembles modeling the architecture directly. Textual model transformation descriptions such as those by ATL are verbose and possibly harder to understand.

The sequence of design decisions captures the evolution of the model. Any intermediate version of the model can be reconstructed by replaying the design decision up to that point. This allows the designer to try out several alternative decisions and inspect the produced architecture. At the same time the design decisions are documented and linked to the model.

6.1 Future Work

There is a need for a large industrial case study. Such a case study can be used to find out if design decisions provide an appropriate granularity for model evolution in practice and with more complex refinements.

The proposed representation for design decisions does not presuppose that the documented decision is actually realized. However, we would need to provide an additional infrastructure for storing design decisions that were rejected.

When the design decision is captured explicitly and to some degree of formality, the decision itself can become the subject of analysis. For example the relation between different design decisions, such as dependency relations, can be calculated automatically. The model transformation provides a precondition (the left-hand side) and a postcondition (the right-hand side) that can be used for deducing dependencies.

Acknowledgements

This work has been partially funded by the ARTEMIS projects CESAR and iFEST.

References

Decoupling Operation-Based Merging from Model Change Recording

Stephen C. Barrett, Patrice Chalin, and Greg Butler
Concordia University, Montreal, Quebec, Canada
{ste_barr, chalin, gregb}@encs.concordia.ca

Abstract. Two general approaches to the merging of models exist: state-based, and operation or history-based. Naturally, using one over the other involves trade-offs. Up to now, a major drawback of the latter approach has been overly tight coupling between the tool performing the merge, and the recorder tracking the changes made to the models. We break this coupling through the use of an abstraction layer, and a tool-independent means of representing model changes: conceivably allowing even cross-tool models to be successfully merged. Further, the design does not preclude the application of state-based methods. The decoupling architecture and techniques are described in the context of their implementation in our model merging tool, Mirador.

Keywords: change recorder decoupling, change metamodel, operation-based merging, state-based merging, model merging, MDD, Mirador

1 Introduction

The importance of model merging to model-driven development (MDD) can be gauged by the literature, which points to a great need for research into, and development of tools that synchronize and merge models [1]. And by the number of conferences and workshops dedicated to the closely related topics of software versioning, model comparison, model differencing, model evolution, and the like. Though accepting of its importance, the community is divided over how to best accomplish model merging. Where one camp maintains that an approach based on model state will be most effective, the other insists that correctness requires making use of change operations recorded over the course of a modeling session.

As with many dichotomies, the approaches are less contradictory than first appears. This was brought sharply into focus for us during redesign of our model merging tool Mirador. Mirador features the use of multiple comparison strategies for establishing model element correspondence, and table-directed merging on a local level. And Mirador is operation-based. Recently, we turned out attention to what is the most frequently cited objection to operation-based merging: the tight coupling between change recording mechanism and merging tool.

In Mirador we have successfully decoupled the two by: (1) abstracting away differences between change recorder outputs; (2) reconstructing modified models from the change logs; and (3) incorporating change history into the models.
themselves. Implied by 2, and made explicit in this paper is that an operation-based stance does not necessarily preclude the use of state-based methods.

The paper starts with a brief look at the two merging approaches, and model difference representation. Mirador and its architecture are presented in Section 3. Recorder/merger decoupling is begun in Section 4 with discussion of change operation abstraction and model reconstruction, and completed in Section 5 which adds the operation abstractions to the model. Related work is covered in Section 6. The final section summarizes the paper and proposes future work.

2 Background

In a cooperative endeavor multiple developers working in parallel inevitably alter the same artifact, which must eventually be reconciled. Under MDD the artifacts to be reconciled are models. It is up to model merging tools then, to bring these diverging replicas (i.e., initially identical copies) into agreement. If the common ancestor of two versions to be merged is available, the merge is three-way, which is used extensively since it can resolve the so-called add-delete problem [2].

2.1 Approaches to Model Merging

When it relies on only a model’s initial and final states, a merge tool is said to be state-based. The schematic on the left side of Fig. 1 shows two modified replicas being merged after extracting their differences. Individual difference outcomes are then selected, one side’s over the other, not only for their contributions to a more complete model, but also for the absence of conflicts and inconsistencies. State-based merging: is in general easier to implement, can only reconstruct a course change history, and is independent of the modeling tool/change recorder.

The operation-based schematic on the right side of Fig. 1, has access to a history of replica changes. Changes are simplified, selected, interleaved, and reordered so as to create a conflict-free merged operational trace, which can then be run against the source model to generate a merged version. Operation-based
merging: has information that can resolve the add-delete problem and some conflicts, and requires tight coupling with the modeling tool/change recorder.

### 2.2 Representing Model Differences

Cicchetti, et al. [3] have proposed capturing the differences between models as a first class artifact through the use of a difference metamodel (MMD) that extends some given metamodel (MM). The MMD (left side of Fig. 2) stipulates the derivation of three new types for each model element type of the MM. With this extended metamodel in place, modifying a model conforming to MM results in a new version conforming to MM and MMD that represents the initial and final states of the model undergoing modification. The Ecore model fragment on the right side of Fig. 2 illustrates its usage. The appearance of MMD elements in the fragment indicate that the model has changed, specifically: class \( c_4 \) and attribute \( a_1 \) have been added, a unidirectional relationship \( r_2 \) inserted between \( c_3 \) and \( c_4 \), and the association \( r_1 \) between \( c_1 \) (now \( c_3 \)) and \( c_2 \) removed.

![Fig. 2. Difference metamodel and a difference model fragment.](image)

The initial model consists of all the **Changed** and **Deleted** MMD elements (\( c_1, r_1 \)), and any MM elements that have not been updated (\( c_2 \)). While the final model consists of all MM elements (\( c_2, c_3 \)), plus the **Added** MMD elements (\( c_4, r_2, a_1 \)). Everything in the final model is reachable from the model’s root node (not shown). The direction of the links between MM and MMD elements however, make portions of the initial model unreachable, necessitating that references to them be maintained elsewhere. Thus tools that are unaware of the MMD see only the final model, while MMD aware tools, provided they are supplied with this reference list, see both.

We note that since the difference metamodel only captures the end states of a modeling session, it cannot by itself support operation-based merging. Also, the difference representation reveals what has changed, but not in precisely what way. This information must be extracted from the model by performing a *diff* between the final version of an element and its initial version. Other things to note are that: adding a new element \( X \) inserts a subclassed **Added** \( X \) object
into the final model; deleting an element $Y$ removes it from the final model, but leaves a “ghost” Deleted$Y$ in the initial model; adding, deleting, or altering a contained element changes its container (e.g., $c1$ due to $a1$); and meta-object names are for human use, being assigned as new objects are created.

3 The Mirador Model Merger

Mirador is an operation-based merging tool being developed at Concordia University. Written in Java, it can run stand-alone, or as a Fujaba plug-in [4]. It uses Ecore to represent changes that have been made to the models to be merged.

3.1 User Interface

Users interact with the tool through three panels. The first panel parses the model change logs. A file containing model element pairings established in a earlier merge session may also be read in. The second panel is concerned with matching the elements of one model with those of the other, either interactively or automatically. Merging based on these pairings, takes place in the last panel.

Conflict identification and merging require a mapping of an element in one model to its corresponding entity in the other. In contrast to most tools which exploit only one matching approach, Mirador makes use of similarity measures obtained from multiple strategies, as well as an overall score computed with a weighted distance function [5]. Up to eight strategies—which may be externally supplied—can be selectively loaded at start-up time. A partial screenshot of the Element Matching Panel is shown in Fig. 3. Seen running across the top of the GUI are fields for altering the weights assigned to each of the loaded evaluators.

![Fig. 3. Mirador displaying similarity measures after comparing model elements.](image-url)
Trees are used to display the left and right versions of the model. This emphasizes element ownership and containment, and has the virtue of simplicity. It also does not conflate changes to the model with changes to its view. A common condition that poses several traps for the unwary [6]. A node’s details are given in the table beneath its tree. Shown here are candidate matches for the selected item, that is, all elements of the same kind found in the other model, ranked by similarity. Grayed out elements have been paired. A pair can be broken up with the **Unmatch** button. Likewise, available elements can be paired with **Match**. Elements can be automatically matched by similarity, with the **Threshold** field specifying the minimum similarity score: Any elements not already matched when the user advances to the Model Merge Panel will be paired with their highest ranked and available candidate that is at, or above the threshold.

### 3.2 Tool Architecture

Mirador’s components and inter-process data flow are informally documented in the diagram of Fig. 4. There are three subsystems, each corresponding roughly to a page of the Mirador wizard. Input in the form of raw model change logs comes from the tracking recorders at the far left. This input stream along with the abstraction layer (MAL) constitute the input subsystem. The comparison subsystem is composed of a comparison unit and an assortment of similarity evaluators. It uses a visualization component which will eventually incorporate the graphic facilities of EMF Compare [7]. The merge subsystem uses planes of local change operations to analyze the merge, and decision tables to drive it.

![Fig. 4. Mirador architecture.](image)

### 4 Change Record Isolation

There are two related, but slightly different aspects to decoupling Mirador from the change tracking mechanism, to be take up in this and the next section. The first has to do with deciphering the recorder-specific change logs, and the second with how model changes are represented to downstream processes.
4.1 Common Change Record Objects

At present, Mirador is limited to working with Fujaba models. Because it is in reality a historical trace of changes, a Fujaba model possesses the characteristics common to other traces, and as such can serve as an exemplar: Typically, changes are logged in a flat file as single-line records, each made up of fields delimited by a special character, with records grouped into transactions. Change logs differ primarily in their change record formats and transaction layouts. To neutralize these differences Mirador adopts a common change record structure.

The raw change records of a particular change recorder are converted into objects of this common structure in passing through that recorder’s dedicated MAL interface. The recorder-specific Fujaba class hierarchy can be seen in Fig. 5 alongside the recorder-neutral Mirador hierarchy, with ellipses alluding to similar hierarchies needed to support other change recorders. Providing tool-specific hierarchies makes for greater modularity, and simplifies debugging.

Each set of tool-specific leaves of the ChangeRecord sub-hierarchy is replaced by a set of Mirador classes that represent only those features of interest for merging. The ChangeRepository class for a tool reads the logs of its change recorder, delegates record creation to its factory, and gathers changes into left and right transactions. A ChangeTransaction serves to bundle the change records together at the model element level; the common version doing so in a recorder-independent fashion.

During record gathering, the repository will eliminate any redundant operations it recognizes. For instance, upon creation, an element may be given a temporary name only to be replaced by a user supplied name at a later point in the same transaction. The temporary operation can be safely discarded. The common repository also acts as a container for the Ecore models that will be eventually reconstructed from the change operations.

Thus two steps have been taken towards decoupling: the first isolates the textual nature of change log records, and the second removes dependencies on
recorder-specific objects. Though adequate for internal use, having to accommodate the Mirador replacement objects places a burden on external actors. Still, this is a great improvement over having to deal with multiple recorder formats.

4.2 Reconstruction of Changed Model Versions

To be truly independent of recorder idiosyncrasies, we need to work with models rather than the object-oriented stand-ins for recorded changes described above. In this spirit Mirador reconstructs Ecore versions of the left and right models by “executing” the operations of the respective change logs.

The changes of a log are executed naively, with little attention paid to underlying semantics. Mirador navigates the changes from the model’s root on down, creating and cobbled elements together as it goes, employing the Ecore API in a fairly straight forward manner. The ease with which existing elements may be manipulated and nested is the main reason behind our selection of Ecore over UML2. The result is an Ecore model structured in terms of containment, that is to say, as parents and children.

At the time of this writing, Mirador knows of packages, classes, operations, attributes, and bidirectional associations. Of these, only associations raise any difficulties. Many modeling tools considers associations to be first class entities, and use a classifier each for the association and its ends. Ecore on the other hand, models a bidirectional association and its roles with two references. The appearance of an association in the model then, requires a translation of three metamodel objects and their properties into two. This is the bidirectional case; a unidirectional association is translated into a single reference.

Having full reconstructions of the left and right models enables downstream processes to perform state-based analyses and manipulations. Any operation-based work undertaken at this point however, would involve using the change operation objects as saved in the MiradorRepository. To achieve total decoupling, these change operations must be somehow “hung off” the elements of the models that they change. The next section proposes a means for doing just that.

5 Capturing Model Evolution

The two Ecore models outputted from the abstraction layer are sufficient for the comparison subsystem to accomplish its goal, but may not satisfy all similarity evaluators. For example, the “by history” evaluator scans the change operations for clues that could aid with matching, and the “by ECL” evaluator runs arbitrary scripts of the Epsilon Comparison Language [8] which could conceivably attempt to examine the change history.

While model comparison in Mirador has a kind of soft dependency on change history, its merging has a hard dependency. These needs can be fulfilled in a tool independent manner by making the changes part of the models to be merged. We use the metamodel of Cicchetti, et al. [3] for this purpose, but as their work is intended to model differences (see Section 2.2) and not change sequences, we have extended it as explained in the next subsection.
5.1 The Changed Metaclass

When reconstructing the models from their change logs, Mirador wraps each Ecore element it creates in an EcoreWrapper. The wrappers provide conveniences like element ID and ancestor details that we would rather not pollute the model with. They act as scaffolding; to be removed once the models have been merged. They also make it possible to extend Ecore with the MMD without having to modify Ecore itself, and to maintain links to all MMD elements.

The iconified model at the top right of Fig. 6a—an abstract class AType contained in package root_pkg—is represented by the wrapped Ecore metaclasses beside it. The icons of 6b show this model evolving through two changes: the class being renamed to LType, and then being made concrete. In the original difference metamodel these changes result in one ChangedEClass object being created with an updated reference to the modified object w2. In our extension of the MMD there are two ChangedEClass objects, one for each change, along with linkage in the opposite direction by which they can be reached from w2.

![Fig. 6. Representation of a model initially (a), and after two changes (b).](image)

Rather than store the original state of the updated element in the Changed metaclass directly, we save it as a nested captured object. Admittedly, not as clean as the original design that replicated the MM element with an added updated reference. It is simply a concession to implementation, where to transcribe an Ecore object into another is quite difficult, but to make a copy is easy.

At this point Mirador has been completely decoupled from change recorders. In fact, any change history dependent tool that can navigate an Ecore model could make use of the abstracted output to decouple itself in the same way.

6 Related Work

merging for managing changes to object-oriented databases. They described a merge as a weave of primitives in a transformation grid, related non-commutating operations to conflicts, and proposed three algorithms for detection. In Mirador we have reoriented their grid, and added a third dimension for matching strategies [12]. Renewed interest in the approach is typified by the works of Koegel, et al. [13], and Schmidt, et al. [14] who argue for transaction based merging.

Properly matching model elements is vital to successful merging, and approaches to it have been varied. Lippe and Oosterom did not concern themselves with entity matching. Others like Pottinger and Bernstein take it as a given [15], and many tools insist on unique identifiers to do any useful work [6]. More robust and flexible ideas are offered by Xing and Stroulia [16] who match based on name and structural similarity, or Kolovos [8] who furnishes a general comparison language. Mirador draws from these and others, to offer an assortment of strategies for comparing and matching. Treude, et al. [17] show how to reduce the inherent $O(n^2)$ complexity of model element comparison to $O(n \log n)$.

Cicchetti, Di Ruscio, and Pierantonio [3] describe a metamodel independent way of representing model differences suitable for state-based merging. In Mirador we have extended their difference metamodel to track model evolution, making it applicable to operation-based merging. Additionally, we have made design changes to facilitate constructing the difference model in Ecore.

7 Conclusion and Future Work

This paper has presented techniques for erasing the dependency that operation-based model merging has on the tools used for recording changes made to models. We also briefly presented our own merging tool Mirador, which incorporates the described techniques, along with novel facilities for matching model elements based on multiple similarity measures.

Much work remains to be done on Mirador. Currently it assumes that the traces passed in to it are complete; that the entire histories of the models in question are contained within the change operations of the logs. This assumption needs to be removed. As part of migrating Mirador to Eclipse, an interface to Eclipse-Change Recorder logs is being added to the abstraction layer. This move will make the visualization capabilities of EMF Compare available to us.

Our work has driven home an important observation worth pursuing: State is always available in an operation-based approach. A full trace contains the ancestor model, the final model, and every model in between. Accordingly, the “either or” question posed by the perceived dualism of state vs. operation based should be replaced with a question of “more or less.” That is, “Is the state-based approach good enough for our needs?” or “Is it worth the extra space, computational (and yes, programming) effort to follow an operation-based approach?”

References

Towards Semantics-Preserving Model Migration

Markus Herrmannsdoerfer and Maximilian Koegel

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
{herrmama, koegel}@in.tum.de

Abstract. Like other software artifacts, modeling languages and thus their metamodels are subject to change. When a metamodel is changed, existing models may no longer conform to it. To avoid loss of information, the existing models need to be migrated. Manual migration is tedious and error-prone, and thus model migration needs to be automated. However, building an automated model migration is a difficult endeavor, as it needs to preserve the meaning of a possibly unknown number of models. In this paper, we extend our operation-based approach to automate model migration by a means to ensure semantics preservation. A coupled operation not only adapts the metamodel and provides the appropriate model migration, but also adapts the semantics definition. Semantics preservation is thus ensured constructively by defining appropriate couples of semantics adaptation and model migration. We showcase the approach using the well-known Petri net example evolution.

1 Introduction

Due to evolving requirements, modeling languages are subject to evolution [1]. A modeling language is evolved by first adapting its metamodel to the evolved requirements. When the metamodel is adapted, dependent artifacts like e. g. editors and interpreters may need to be co-adapted. Most importantly, existing models may no longer conform to the adapted metamodel. To be able to use the existing models with the evolved modeling language, they have to be migrated to conform to the adapted metamodel. Manual migration is tedious and error-prone, and thus model migration needs to be automated [2].

To automate model migration, we follow an operation-based approach [3]: When adapting the metamodel, the model migration is recorded as a sequence of coupled operations in an explicit history model [4]. A coupled operation encapsulates both the metamodel adaptation as well as the reconciling model migration. Model migration can be further automated by reusing recurring combinations of metamodel adaptation and model migration. Through a number of case studies, we have shown that most of the model migration in practice can be covered by reusable coupled operations [2, 3, 5]. The model migration needs to be defined manually only in case there is no appropriate reusable coupled operation available. We have implemented this operation-based approach for the Eclipse
Modeling Framework (EMF) [6] in a tool called COPE (COuPled Evolution of metamodels and models)\(^1\).

In addition to automating the model migration, it has to be ensured that the model migration preserves the meaning of the existing models. If the meaning is not preserved, important information may be lost during the automated model migration. This problem is further aggravated by the fact that we need to migrate a possibly unknown number of models. Of course, we can only talk about semantics preservation, in case the semantics of a modeling language is explicitly defined. In practice, the semantics of a modeling language is usually defined by means of a simulator or code generator.

In this paper, we extend our operation-based approach by a means to ensure semantics preservation. A reusable coupled operation not only adapts the metamodel and provides the appropriate model migration, but also adapts the semantics definition. Semantics preservation is thus ensured constructively by defining appropriate couples of semantics adaptation and model migration. We showcase the approach using the well-known Petri net example evolution [7]. For custom coupled operations which require manual specification of the model migration, the semantics definition also has to be adapted manually. Since custom coupled operations are rather rare in practice according to our case studies [2, 3, 5], it is sufficient to manually prove semantics preservation in these cases.

The paper is structured as follows: Sec. 2 presents how we assume that both the syntax and semantics of a modeling language are defined. Sec. 3 introduces our operation-based approach to build a semantics-preserving model migration. Sec. 4 demonstrates the approach using the well-known Petri net example evolution. Sec. 5 discusses related work, and Sec. 6 concludes the paper.

2 Definition of Modeling Languages

_Modeling Languages._ Before we can characterize the evolution of a modeling language, we have to formally define our understanding of a modeling language.

Definition 1 (Modeling Language). A modeling language \( \mathcal{L} = (\mathcal{A}, \mathcal{S}) \) is a tuple of abstract syntax \( \mathcal{A} \) and semantics \( \mathcal{S} \).

Usually, modeling languages also define a concrete syntax to represent models in a human-readable manner. However, the concrete syntax which may also have to be considered for modeling language evolution is out of this paper’s scope. Here, we assume that the abstract representation of a model contains all the information necessary to determine its semantics.

Definition 2 (Abstract Syntax). The abstract syntax defines the (possibly infinite) subset \( \mathcal{A} = \{m_1, m_2, \ldots\} \subseteq \mathcal{M} \) of the models that are syntactically correct. \( \mathcal{M} \) defines the set of all possible models which are essentially labeled and attributed graphs.

\(^1\) see COPE web site, http://cope.in.tum.de
**Definition 3 (Semantics).** The semantics $S = (SD, SM)$ is a tuple of semantic domain $SD$ and semantic mapping $SM$. The semantic domain $SD$ specifies the concepts that exist in the universe of discourse, and the semantic mapping $SM : A \rightarrow SD$ interprets a model from $A$ w.r.t. the semantic domain $SD$.

The semantic mapping needs to be total so that every syntactically correct model $m \in A$ has a semantic interpretation $S(m)$. In the following, we require the notion of semantic equivalence $\equiv$ on the semantic domain. Two models $m_1, m_2 \in A$ are semantically equivalent, if the interpretation returns equivalent objects from the semantic domain, i.e. $S(m_1) \equiv S(m_2)$.

**Definition of Modeling Languages.** Abstract syntax and semantics of a modeling language are also defined by means of models using special modeling languages.

**Definition 4 (Metamodel).** A metamodel $mm \in MM \subset M$ defines the syntax of a modeling language. The modeling language to define metamodels has the semantics $E : MM \rightarrow \mathcal{P}(M)$ which maps a metamodel $mm \in MM$ to the set of syntactically correct models $E(mm)$.

**Definition 5 (Semantics Definition).** A semantics definition $sem \in SEM$ defines the semantics of a modeling language. The modeling language for semantics definitions has the semantics $I : SEM \rightarrow S$ which maps a semantics definition $sem \in SEM$ to a semantics $I(sem)$.

There are different modeling languages to define metamodels as well as different ways to define semantics, e.g. operational, denotational, axiomatic. In this paper, we define the abstract syntax by means of UML class diagrams, similar to MOF (Meta Object Facility), and the semantics as operations on metamodel classes, similar to Kermeta [8].

**Running Example.** In this paper, we use the well-known Petri net evolution [7] as a running example. Fig. 1 shows the metamodel, and Fig. 2 the semantics definition of the first version of the Petri net modeling language. To define the semantics, we use a notation that is similar to the Groovy scripting language. The first line of each block defines the signature of an operation which is associated to a metamodel class. Within the block, all accesses are performed on an object of the respective metamodel class.

![Fig. 1. Metamodel of the Petri net modeling language version 1.](image-url)
Net.run(Interaction i):
    reset()
    ts = getActivatedTransitions()
    while (!ts.isEmpty()) {
        if (ts.size() == 1) ts.get(0).fire()
        else i.choose(ts).fire()
        ts = getActivatedTransitions()
    }

Interaction.choose(List<Transition> ts):
    ...

Net.reset():
    places.each{ p -> p.reset()}

Net.getActivatedTransitions():
    List<Transition>
    return transitions.collect{ t -> t.isActivated()}

Transition.isActivated(): boolean
    return src.every{ p -> p.isActivated()}

Transition.fire():
    src.each{ p -> p.decrement()}
    effect.execute()
    dst.each{ p -> p.increment()}

Place.reset():
    current = initial

Place.isActivated(): boolean
    return current >= 1

Place.decrement():
    current = current - 1

Place.increment():
    current = current + 1

**Fig. 2.** Semantics definition of the Petri net modeling language version 1.

Petri Nets consist of places and transitions. A Place has a number of tokens (current) which are reset to the number of initial tokens when the net is started. A Transition transfers tokens from src to dst places. A transition isActivated if every src place has at least one token. If multiple transitions are activated at the same time, user Interaction is required to choose one. When a transition fires, the tokens of the src places are decremented, its effect is executed and the tokens of the dst places are incremented. The semantics definition thus maps Petri net models to traces of action executions.

### 3 Evolution of Modeling Languages

*Modeling Language Evolution.* A modeling language is evolved by adapting the definition of the syntax and/or semantics of the modeling language.

**Definition 6 (Modeling Language Evolution).** A modeling language is evolved by adapting the metamodel from mm₁ to mm₂, and the semantics definition from sem₁ to sem₂. The metamodel adaptation changes the syntax of the modeling language from $E(mm₁) = M₁$ to $E(mm₂) = M₂$. The adaptation of the semantics definition changes the semantics of the modeling language from $I(sem₁) = S₁$ to $I(sem₂) = S₂$, with $Sₙ = (SDₙ, SMₙ : Mₙ → SDₙ)$ for $n = 1, 2$.

Note that the semantic domain may change, when a modeling language is evolved. Therefore, the semantic equivalence $≡$ needs to be extended to work on models from different semantic domains, which may require abstraction.

**Example.** The Petri net modeling language needs to be evolved to cater for weights that are different from 1 [7]. Fig. 3 shows the adapted metamodel, and Fig. 4 the adapted semantics definition.
Arcs are introduced to define the incoming and outgoing weights for transitions. PTArcs define the number of tokens by which the src places of transitions are decremented. TPArcs define the number of tokens by which the dst places of transitions are incremented.

**Model Migration.** We have to migrate models that are no longer part of the evolved modeling language or whose semantics have changed. Moreover, we can define certain preservation properties for model migration.

**Definition 7 (Model Migration).** Model migration is required if \( \exists m \in \mathcal{M}_1 : m \notin \mathcal{M}_2 \vee \mathcal{S}_1(m) \neq \mathcal{S}_2(m) \). Model migration is defined as a function \( \text{mig} : \mathcal{M} \rightarrow \mathcal{M} \) that maps a source to a target model.

**Definition 8 (Preservation Properties).** A model migration \( \text{mig} : \mathcal{M} \rightarrow \mathcal{M} \) is syntax-preserving if \( \forall m \in \mathcal{M}_1 : \mathcal{S}(m) = \mathcal{S}(\text{mig}(m)) \). A syntax-preserving model migration is semantics-preserving if \( \forall m \in \mathcal{M}_1 : \mathcal{S}(m) = \mathcal{S}(\text{mig}(m)) \).

**Coupled Operations.** Until now, coupled operations only encapsulate metamodel adaptation and model migration and thus can only ensure syntax preservation.
Definition 9 (Syntax-preserving Coupled Operation). A syntax-preserving coupled operation provides metamodel adaptation \( \text{adm} : \mathcal{MM} \to \mathcal{MM} \) and model migration \( \text{mig} : \mathcal{M} \to \mathcal{M} \). It is syntax-preserving for a metamodel \( \mathcal{mm} \in \mathcal{MM} \) if \( \forall m \in \mathcal{E}(\mathcal{mm}) : \text{mig}(m) \in \mathcal{E}(\text{adm}(\mathcal{mm})) \).

To also ensure semantics preservation, a coupled operation needs to be extended to also encapsulate the adaptation of the semantics definition.

Definition 10 (Semantics-preserving Coupled Operation). A semantics-preserving coupled operation is a syntax-preserving coupled operation that also provides an adaptation \( \text{ads} : \mathcal{SEM} \to \mathcal{SEM} \) of the semantics definition. It is semantics-preserving for a metamodel \( \mathcal{mm} \in \mathcal{MM} \) and semantics definition \( \mathcal{sem} \in \mathcal{SEM} \) if \( \forall m \in \mathcal{E}(\mathcal{mm}) : I(\mathcal{sem})(m) \equiv I(\text{ads}(\mathcal{sem}))(\text{mig}(m)) \).

Example. The coupled operation Reference to Class that is required for the Petri net evolution replaces a reference \( r \) by an explicit reference class \( R \). The metamodel adaptation which is shown in Fig. 5 makes the reference \( r \) composite and creates the reference class \( R \) as its new type. Single-valued references \( s \) and \( t \) are created in the reference class \( R \) to target the source and target class \( C_1 \) and \( C_2 \) of the original reference \( r \). The model migration replaces links conforming to the reference by objects of the reference class, setting source and target reference appropriately. Now, the coupled operation is already syntax-preserving.

![Fig. 5. Metamodel adaptation of the coupled operation Reference to Class.](image)

To make it semantics-preserving, we also need to define an appropriate adaptation of the semantics definition. All accesses to \( r \) on instances of \( C_1 \) need to be replaced by \( r.\text{collect}(b \to b.t) \). Similarly, all accesses to the opposite reference \( o \) on instances of \( C_2 \) need to be replaced by \( o.\text{collect}(b \to b.s) \).

4 Case Study for Coupled Evolution

In this section, we use a selection of semantics-preserving coupled operations to evolve the Petri net modeling language from version 1 (see Fig. 1 and 2) to version 2 (see Fig. 3 and 4).

Reference to Class. To introduce the reference classes \( \text{PTArc} \) and \( \text{TPArc} \), the coupled operation Reference to Class is applied twice. Fig. 6 illustrates the metamodel adaptation, and Fig. 7 the adaptation of the semantics definition. Changes are highlighted in red and by dashed boxes and lines. The model migration introduces reference objects for all links of the references \( \text{src} \) and \( \text{dst} \).
Refactoring of the Semantics Definition. To simplify the semantics definition, we can refactor it by folding the collect into the each and every statement. The result of this refactoring is shown in Fig. 8.

Rename. To facilitate understanding the metamodel, a number of references need to be renamed. Fig. 9 illustrates the metamodel adaptation and highlights which references are renamed to a new name. To be semantics-preserving, the links need to be renamed accordingly in models, and the accesses to these references need to be renamed accordingly in the semantics definition (see Fig. 10).

Drop Composite. PTArcs and TPArcs should not be contained by the Places from which they originate, but by the Net. As is shown in Fig. 11, Drop Composite drops the composite constraint on the references and creates composite references in Net to contain the PTArcs and TPArcs. The model migration needs to ensure
that the PTArcs and TPArcs are contained by the Net. No adaptation of the semantics definition is necessary, as the existing accesses lead to the same result.

**Extract Superclass.** A common superclass Arc for PTArc and TPArc needs to be created. The metamodel adaptation is shown in Fig. 12. Neither model migration nor adaptation of the semantics definition is necessary, as *Extract Superclass* does not affect existing objects or accesses.

**Refactoring of the Semantics Definition.** To prepare the introduction of explicit weights, we need to refactor the semantics definition as shown in Fig. 13. The operations *isActivated*, *decrement* and *increment* are moved from Place to PTArc or TPArc. The accesses to the operations and to *current* have to be adapted accordingly.

```java
Transition.isActivated(): boolean
   return in.every(pt -> pt.isActivated());
Transition.fire():
in.each(pt -> pt.decrement());
effect.execute();
out.each(tp -> tp.increment());

PTArc.isActivated(): boolean
   return src.current >= 1
PTArc.decrement():
   src.current = src.current - 1
PTArc.increment():
   dst.current = dst.current + 1
```

Fig. 13. Semantics definition after refactoring.
New Attribute. The last step is to introduce the attribute weight as shown in Fig. 3 and 4. To allow for weights different from 1, the semantics definition has to be manually adapted to use the value of weight. By choosing 1 as a default value for weight, no model migration is needed and the semantics is preserved.

5 Related Work

Rose et al. classify approaches to automate the migration of models into manual specification, operation-based and matching approaches [9].

Manual specification approaches provide languages tailored for model migration to manually specify the migration. Sprinkle and Karsai present a graph transformation language that requires to specify the migration only for the metamodel difference, automatically copying unaffected model elements [10]. Narayanan et al. present MCL (Model Change Language) to specify the metamodel changes which can be used for both model migration [11] and model transformation adaptation [12]. When specifying the semantics as a model transformation, MCL can be used to perform semantics-preserving model migration. Flock also automatically unsets model elements which are no longer syntactically correct [13]. However, the manual specification approaches do not provide explicit support to ensure that the specified migration is semantics-preserving.

Operation-based approaches specify the model migration as a sequence of coupled operations which encapsulate metamodel adaptation and model migration. Wachsmuth classifies a set of coupled operations according to semantics and instance preservation properties [7]. However, Wachsmuth uses the term to denote the preservation of the metamodel’s semantics, i.e. the set of syntactically correct models, rather than the preservation of the modeling language’s semantics, i.e. the function that maps each model to its meaning. Our approach COPE extends Wachsmuth’s approach by a means to manually specify custom coupled operations and to define new reusable coupled operations [3]. We have shown that model migration cannot be automated in certain cases, when taking the semantics of a modeling language into account [14].

Matching approaches try to detect a model migration based on the matching between two metamodel versions. Gruschko et al. classify primitive metamodel changes into non-breaking, breaking resolvable and breaking non-resolvable and envision to automatically detect a model migration for breaking resolvable changes [15]. Moreover, Cicchetti et al. are able to detect complex changes in the difference between two metamodel versions [16]. Garcés et al. present a matching language that allows the user to customize the matching process and to add new matching patterns [17]. However, the matching approaches all work only on the metamodel and thus are not able detect a semantics-preserving model migration.

6 Conclusion

In this paper, we have illustrated an operation-based approach to ensure that model migration is semantics-preserving. We have extended coupled operations
that encapsulate metamodel adaptation and model migration by an adaptation of the semantics definition. To get a first impression on the feasibility of the approach, we have successfully applied it to the Petri net example evolution [7]. As future work, we plan to extend our tool COPE by a means to specify the adaptation of the semantics definition to be able to conduct a case study.

Acknowledgments. We thank Elmar Juergens for helpful suggestions. This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grants “SPES2020, 01IS08045A” and “Quamoco, 01IS08023B”.

References

Semi-Automated Correction of Model-to-Text Transformations

Gábor Guta\(^1\), András Pataricza\(^2\), Wolfgang Schreiner\(^1\), and Dániel Varro\(^2\)

\(^1\) Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria
\{Gabor.Guta, Wolfgang.Schreiner\}@risc.uni-linz.ac.at
\(^2\) Department of Measurement and Information Systems, Budapest University of Technology and Economics
\{pataric, varro\}@mit.bme.hu

Abstract. Model-to-text (M2T) transformations are critical in Model Driven Software Development (MDSD) to automatically derive product quality source code of a target application. Template based code generators and XSLT are the most popular industrial ways to implement such M2T transformations. However, debugging and correcting a M2T transformation can be time consuming for developers, even in case of minor problems. Our goal is to support the development of M2T transformations by providing automatic corrections if the developer introduces minor changes into the result of a transformation. In this paper we describe a by-example approach for the automatic correction of M2T transformations.

1 Introduction

In Model Driven Software Development (MDSD), model-to-text (M2T) transformation languages (such as XSLT, Velocity, JET, Xtend, etc.) frequently serve as means for automated code generation in order to efficiently deliver high-quality software. However, due to the steep learning curve of developing custom M2T transformations, still relatively few software engineers master such technologies. Furthermore, current industrial practice of MDSD focuses mostly on the production of software with the help of out-of-the-box transformations while the development of new transformations [1] is somewhat ad hoc.

In an industrial context, M2T transformations are usually developed in an iterative way where a transformation expert delivers a first version, which is iteratively tested by using sample inputs obtained from future users (i.e. regular software engineers) of the code generator. The automatically derived output is then compared to the expected output, and the M2T transformation is manually corrected afterwards by the transformation expert.

However, industrial practice shows that in many cases, there are only minor differences between the expected output and the derived output and the corrections of the M2T transformations are almost trivial. Our approach was motivated by the de facto experience of the first author in industrial projects described in [9,
where a complex ASP.NET application (approximately 200 K lines of code) was generated by XSLT scripts. In case of such complex projects, we found that even the localization of simple errors in XSLT scripts is already very time consuming. Furthermore, in our projects the developers at the company had almost no experience with transformation languages. Small sized companies having similar background play important role in software development business (e.g. the average size of software companies were smaller than 10 people per company in Austria [12]). Thus any kind of assistance in developing M2T transformations would count in practice.

Our aim in the paper is to reduce effort needed to develop M2T transformations by (1) supporting regular developers to introduce changes in the result (output) of the M2T transformation, and (2) providing semi-automated correction techniques for the M2T scripts when developers introduced only minor, syntactic changes in the result of the transformation. Intuitively, modifications are classified as simple (point-like) textual changes of the output which does not necessitate changes in the control flow of the M2T transformation. As a consequence, the main contribution of the paper is to infer minor, but still significant modifications (changes, additions, deletions) of the M2T transformation script from the modification of the output.

In Section 2 we provide an overview of our approach. Then in Section 3 we give a demonstrative example and according to it we describe each step in detail. In Section 4 we discuss the related work and in Section 5 we conclude.

2 Overview

In this section we give an overview of our system which is capable to automatically infer minor modifications of the transformation. In Figure 1 we depict the architecture of the system. The rectangles represent the processed or produced artifacts: the green (gray) ones are the inputs, the yellow (light gray) ones are the intermediate data, and the red (dark gray) one is the result. The rounded rectangles represent the components of the system.

To implement such a system we assume that the selected M2T transformation language makes it possible to extract the control flow graph of the M2T transformation statically. The interpreter or the compiler is also expected to provide some way to record the trace of the execution. For the output language specification we need the regular grammars of the lexer to tokenize the output.

We present a system which supports XSLT as the M2T transformation language and T-SQL as the output language. This system takes three inputs: the model, the M2T transformation and the modified output (i.e. the modifications of the generated output) and produces a list of suggested transformation modifications.

1. The control flow graph \( (CFG) \) of the \( M2T \) transformation is first built by the \( CFG \) builder (Section 3.1).
2. Now the transformation engine executes the transformation to generate the output and to save the trace (Section 3.2).
3. After the user modifies the output, both the generated and the user-modified outputs are tokenized by the lexer to achieve a meaningful granularity for the comparison. The output comparison is done by a standard algorithm to produce a sequence of the tokens in which the differences are marked (diff sequence in short). The additions to the generated output and deletions from the generated output can be mapped to the CFG with the help of the trace (Section 3.3).

4. The CFG is abstracted to an annotated CFG, which contains only the output instructions and the list of modifications at all nodes and edges. Thus, the CFG annotator carries through two steps: pre-processes the CFG by creating an abstract CFG and the trace is modified according to this abstraction (Section 3.4) and annotates the abstract CFG by assigning modification tables to the nodes and edges according to the trace and the diff sequence (Section 3.5).

5. The inference algorithm processes the lists of modifications stored in the annotated CFG and proposes a list of transformation modifications if the modifications are consistent. If the modifications are contradictory, the algorithm displays the modifications which cause the contradiction (Section 3.6).

To use our system for a different transformation and target languages, the CFG Builder and the Lexer have to be modified, and trace recording of the Transformation Engine has to be configured. These modifications and configurations are usually not the task of programmer, but available from the tool developer.

3 The Auto-Correction Process Step-by-Step

In this section we focus on the description of how minor modifications can be located in the transformation script with high precision by exploiting the trace information to map the modification of the output stream to the control flow graph of the M2T transformation. We omit the detailed description of the inference algorithm, due to the space limitation.
To explain the process of locating minor modifications we consider a very simple example inspired by the transformations developed in our industrial project [10]. The Figure 2 shows (A) a sample XML fragment of a database table description which is transformed into T-SQL code (C) by an XSLT transformation (B). The T-SQL code is a fragment from a database code of an enterprise application, which drops the old stored procedure and creates a new one. The stored procedure returns a single row of the database table whose ID is equivalent to the input parameter of the stored procedure. We assume some minor errors in the transformation: the code emits a semicolon after the database column names instead of a comma, and the "id" suffix is omitted from the procedure name (C). We correct these errors in the example T-SQL code and our goal is to derive the corrected XSLT template (D).

Our system can cope with much more complex transformations to handle (localize or fix) modifications such as (1) inserting complete text fragments in the M2T transformation; (2) localizing the printing of incorrect values taken from wrong model data (e.g. from wrong column); (3) customizing transformations to similar (but slightly different) output formats (ongoing work).

3.1 Extraction of the Control Flow Graph

First, the Control Flow Graph (CFG) is extracted from the transformation program. In Figure 3/A the CFG of the part of the example (Figure 2/B from line 11 to 14) is shown. The nodes are locations in the source code (line/column numbers of the first character of the following instruction) and the edges represent instructions. The source node, the target node and the instruction itself are
stored in the CFG. Control flow related instructions are represented by multiple edges which are marked by no operation (NOP) labels. The CFG has to be derived in a similar way as the transformation engine executes the script, otherwise it cannot be correlated with the trace. A for-each loop and an if instruction can be seen in the Figure 3/A. The for-each loop contains three print instructions, before the if instruction: the first one prints the " " constant, the second and third one print variables. The if instruction contains a print instruction too.

3.2 The Transformation and the Recording of the Trace

As a next step, the M2T transformation is executed on the model to generate the output. The transformation is executed by an existing M2T engine (e.g. XSLT engine). We assume that (1) the transformation is free from run-time errors and (2) the output can be tokenized by the lexer of the output language (but not necessarily parsed by the grammar of the output language). In this step, an execution trace is also recorded containing the location of the executed instructions, the instructions themselves, and the produced output (if any).

The part of such a trace which corresponds to the example shown in Figure 2/B (from line 11 to line 14) is as follows:

(11/6, 11/37, xsl:for-each, -), (11/37, 12/6, constant, " "),
(12/6, 12/35, xsl:value-of select="\$name", "AdrAssignee"),
(12/35, 13/1, xsl:value-of select="@name", "StartDate"),
(13/1, 13/36, xsl:if, -), (13/36, 14/4, constant, ","),
(14/4, 14/10, NOP, -), (14/10, 11/6, NOP, -),
(11/6, 11/37, xsl:for-each, -), (11/37, 12/6, constant, " "),
(12/6, 12/35, xsl:value-of select="\$name", "AdrAssignee"),
(12/35, 13/1, xsl:value-of select="@name", "EndDate"),
(13/1, 14/10, NOP, -), (14/10, 11/6, NOP, -), ...

The trace fragment shows how the elements in Figure 3/A from line 12 to line 17 is processed and output can be seen in Figure 3/C from line 12 to line 13 is generated. The first tuple in the fragment of the trace represents an xsl:for-each
instruction (located in Line 11 Columns 11 to 37), which produces no output. The second tuple in the trace fragment represents a constant print instruction (Line 11 Column 37 to Line 12 Column 6), which emits “” (actually produces ”/n”, but we represent this output with three space in the rest of the examples to save space). This trace is represented graphically in Figure 3/B. The nodes are shown only to make it easier to understand the correspondence to the CFG represented in part A of the figure. The actual trace is the sequence of the edges labeled by the produced strings or the results of expression evaluations.

3.3 Tokenizing and Comparing the Text Outputs

After the modification of the generated output by the user both the generated and the user-modified output are tokenized and compared. Instead of a naive approach based on character-wise comparison of the two outputs, we compare the tokens in accordance with the lexical syntax of the target language. For the token sequence comparison algorithm, a slightly modified version of the Ratcliff/Obershelp pattern recognition algorithm [13] is used. Note, however that the whitespace and comment elements of the output language are also kept and tokenized.

In Figure 4 the output fragment (Figure 3/B) can be seen in various processing phases. The first line shows output strings of the trace. The second and the third lines represent the tokens of the generated output and the user-modified output, respectively. The differences between the output strings of the trace and the tokens of the generated output are: the trace contains dummy elements representing no output “-” and they are tokenized in a different way. The last line shows the diff sequence in which the addition and removals marked with + and -, respectively. The diff sequence is the result of the token comparison of the generated and modified outputs after some post-processing. In the example, the post-processing means that the ”AdrAssigneeStartDate” (”AdrAssigneeEndDate”) token is sliced in two tokens ”AdrAssignee” and ”StartDate” (”AdrAssignee” and ”EndDate”) according to the tokenization in the trace.

![Fig. 4. Fragments of the Trace, the Outputs, and the Diff Sequence](image)
3.4 Preprocessing of the Control Flow Graph and the Trace

To prepare the annotation process, an abstraction of the CFG is created. The edges of the CFG represent instructions, which can be divided into two groups according to the fact whether they produce output or not. The constant string output and the value-of instructions (printing the value of an expression) are output instructions. The control instructions (if, for-each, NOP, etc.) and variable assignment instructions are non-output instructions. The abstract CFG is created from the CFG by merging the nodes connected by edges representing non-output instructions, thus we basically remove instructions which produce no output.

![Fig. 5. Abstraction of the Control Flow Graph](image)

Figure 5/A shows the original CFG and Figure 5/B shows the abstract CFG. The mapping between the two representations is recorded, and marked with gray arrows. For instance, node 12/6 of the original CFG is mapped to node 2 in the abstract CFG as both the incoming and the outgoing instructions print a textual output (a whitespace and the value of variable `name`, respectively). On the other hand, nodes 11/6, 11/37, 13/1, 13/36, 14/1 and 14/10 are merged into Node 1. From this example one can observe that the abstract CFG represents the relevant information for locating the position of the addition/removal in a more compact way than the CFG does.

This abstraction can be carried out as the location of the additions and deletions can be identified in the context of the output instructions. To demonstrate this principle, imagine that a print of a constant expression is inserted before or after the edge between 13/1 and 14/10 nodes (else branch of the xsl:if expression). If the trace contains no information from the execution of the if branch, no change in the behavior with respect to the output of the program can be observed between the two variants.

3.5 Annotating the Control Flow Graph with the Modification

The CFG annotator converts the information from the diff sequence into the difference tables of the edges and nodes of the abstract CFG with respect to a
specific trace (code generation run). This annotation marks if a certain abstract CFG node or edge is represented in the trace or not. If it is represented then we also mark whether is left unmodified, or new tokens are inserted, or existing ones are removed.

The output instruction which produced a specific token can be identified by simultaneously processing the diff sequence and the trace itself. As soon as the instruction which produced (or removed) the token is known, then the position of the token can be located in the abstract CFG.

In Figure 6 the annotation tables for Node 1 are shown, which is generated from the trace of Figure 2/B (together with an extract of the abstract CFG node itself). The node and the edges are labeled the same way as can be seen in Figure 5/B; the XSLT instructions (edges) are additionally marked with letters. The trace elements which traverse Node 1 are represented by (numbered) gray arrows. The connection between the figures can be understood as follows (each line is started by the number of the trace in Figure 6, followed by the corresponding element numbers in Figure 4/Diff Sequence, and closed by the informal description of the trace elements):

- 1 (token 1): Node 1 is passed for the first time, when the control flow reaches the fragment (A) and then a whitespace " " is printed (B);
- 2 (tokens 3-4): Node 1 is passed for the second time, when the value of the @name attribute is printed (C) and then a semicolon ";" is printed (D);
- 3 (tokens 4-5-6): Node 1 is passed for the third time, when the ";" is printed (D) and then a whitespace " " is printed (B);
- 4 (token 8): Node 1 is passed for the last time, when the value of the @name attribute is printed (C) and the control flow leaves the fragment (E).

The elements 4 and 5 of the diff sequence represent differences between the generated output and the user-modified output. The element 4 is stored in the edge table of the Edge D and the element 5 is stored in the node table of Node 1.

### 3.6 Inferring the Modifications

In this step the possible transformation modifications are inferred from the difference tables in the annotated CFG. Then the edit script which updates the
M2T transformation is assembled according to the possible modifications. In Figure 7 the inferred modifications and their conversion to edit script lines can be seen. The figure represents the same fragment which can be seen in Figure 6. The tables belonging to the edges represent whether the edges have to be left unchanged (B, C) or have to be modified (D). The table belonging to the Node 1 shows that a new print token instruction has to be inserted at the end of Edge D.

![Fig. 7. Inferring Modification from the Annotated CFG](image)

The suggested transformation modifications (edit script) of the described example is the following: add token "select_by_id" at 3/48, remove token "select_by" at 3/48, add token "," at 13/37, and remove token ";" at 13/37. The mapping from the possible modifications table to the suggested transformation modifications of the last two elements can be seen in Figure 7.

4 Related Work

The idea of model transformation by example (MTBE) has been recently recognized by the model transformation community [2, 14]. In MTBE, model-to-model transformations are inferred by various methods from a prototypical set of examples. The main difference in our approach is that we require an approximate solution from the developer which is automatically or semi-automatically modified to achieve the required result. XML transformation by example is a conceptually similar problem, but operates over the domain of tree structures instead of graphs [7].

Research results on the automatic correction of the data structures can be found in [4]. The closest work related to our approach is [5], in which a dynamic tainting technique is described to locate errors in the input T2M transformation. This technique adds taints marks to the input which are traced to localize the input data triggers the incorrect output. In case of unknown structures of the examples, the schemas (grammars) can be inferred from a set of examples [8]. A decent overview of inferring automata and transducers can be found in [11]. Although, we have not found any theoretical work on the inference of grammar modification by example, applications can be found which use heuristics to solve such problems. Typical applications are learning dialects of a programming language [6] or error tolerant parsing [3].
5 Conclusions and Future Work

In this paper we presented a technique to semi-automatically infer minor modifications of M2T transformations from modifications of the derived output. A proof-of-concept prototype was implemented in Java and in Python to carry out the automatic correction of XSLT transformations in case of minor modifications. Furthermore, initial experiments have been carried out using the context of a previous industrial project [9], but a more thorough experimental evaluation is an ongoing work.

References

Representation and Visualization of Merge Conflicts with UML Profiles*

Petra Brosch¹, Horst Kargl³, Philip Langer², Martina Seidl¹, Konrad Wieland¹, Manuel Wimmer¹, and Gerti Kappel¹

¹ Business Informatics Group, Vienna University of Technology, Austria
{lastname}@big.tuwien.ac.at
² Department of Telecooperation, Johannes Kepler University Linz, Austria
{firstname.lastname}@jku.at
³ SparxSystems, Austria
{firstname.lastname}@sparxsystems.at

Abstract. The urgent demand for optimistic version control support for software models induced active research within the modeling community. Recently, several approaches have been proposed addressing the task of detecting conflicts when merging two concurrently changed versions of a model. In this context, the holistic representation and supportive visualization of detected merge conflicts pose a challenge. In this paper, we present a modeling language independent conflict model comprising all necessary information to profoundly represent merge conflicts. From this conflict model, we leverage the dynamic extension power of UML profiles by introducing a dedicated conflict profile to visually assist modelers in resolving merge conflicts of UML models. As a result, modelers may resolve conflicts in the concrete graphical syntax conducting their familiar UML editors without tool extensions.

Key words: model versioning, conflict visualization, UML profile

1 Introduction

Like traditional program code, software models are not resistant to change, but evolve over time by steadily undergoing extensions, corrections, and updates. Especially in the context of model-driven engineering (MDE) models are not used for mere documentation purposes only. Instead, models are leveraged as first-class development artifacts. Hence, models are subject to continuous evolution requiring adequate techniques to manage the development process in general and to support the collaborative creation and modifications in particular [8, 17].

The application of version control systems (VCS) is one important way to improve cooperation in software development [6]. Following the optimistic versioning paradigm, every modeler works independently from other team members

* This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation, and Technology and the Austrian Research Promotion Agency under grant FIT-IT-819584 and by the fFORTE WIT Program of the Vienna University of Technology and the Austrian Federal Ministry of Science and Research.
on her personal local copy. When merging the isolated updates into one common version, conflicts might be raised due to divergences in the different replicas triggering the conflict resolution process necessary to obtain a consolidated and reconciled merged version of the modified artifacts [13].

To provide collaborative modeling support, text-based versioning systems successfully applied for the versioning of code such as Subversion\(^4\) and CVS\(^5\) have been reused. It has been quickly realized that XMI serializations are not the appropriate representation for detecting and resolving conflicts between concurrently edited model versions, because modelers are familiar with the concrete graphical syntax but not with computer internal representations. Thus, some dedicated approaches have been proposed for visualizing differences of UML models. They construct a dedicated view using the concrete syntax, which combines and highlights changes of both models using coloring techniques [12, 15]. Hence, the modeler remains in her familiar modeling environment. However, these approaches require the implementation of special editor extensions.

In this paper we pursue this idea by proposing an approach for representing and visualizing merge conflicts for UML models based on UML profiles. As a conceptual basis, we first present a holistic and language independent model for representing conflicts. From this model, we derive a dedicated conflict diagram view with additional annotations for marking changes and conflicts between UML models. These annotations are defined by a UML profile, which allows on the one hand, to display annotated models in arbitrary UML editors without requiring any tool extensions. On the other hand, it yields a specification for tool vendors to integrate mechanisms for user friendly conflict resolution and filtering support. This additional information is valuable for the final merge process.

Starting with a brief motivating example in Section 2, we continue with discussing how to represent conflicts emerging from two concurrently modified versions of one model in Section 3. In Section 4, we present a visualization of conflicts using UML profiles. Finally, in Section 5 we conclude and give an outlook on future work.

## 2 Motivating Example

Harry and Sally check out the UML Class Diagram V0 depicted in Fig. 1 from a central repository and concurrently perform several modifications.

Sally renames the class Element (C1) to Shape and sets this class abstract. Next, she performs the refactoring pullUpField by removing the common attribute area from all subclasses Square (C2), Circle (C3), and Line (C4) and adding it to the common superclass Element (C1). Furthermore, she adds two new attributes to the class Circle (C3), namely perimeter and radius. Finally, the class Square (C2) is set as superclass of Rectangle (C6). The result of Sally’s work is depicted as V0’ in Fig. 1, which she commits to the repository.

\(^4\) http://subversion.tigris.org
\(^5\) http://cvs.nongnu.org
In parallel, Harry removes the class Circle (C3) as well as the attribute area from the class Line (C4). Next, he introduces a new class Point (C5) which is set as subclass of Element (C1). In addition, he-renames Element (C1) to Figure. Finally, Harry sets the class Rectangle (C6) as superclass of the class Square (C2).

When Harry tries to check in his modifications, several conflicts are reported (cf. right part of Fig. 1). Only the operation update(C1.isAbstract, true) may be merged unproblematically (depending on the unit of versioning). The refactoring pullUpField(C1, area) is conflicting with both, the introduction of the class C5 and the deletion of the attribute area of the class C4. The precondition of this refactoring is violated, because on the one hand the attribute area in the class C4 is deleted and on the other hand the added class C5 does also not provide the required attribute. Since both, Harry and Sally, have renamed the class Element, an update/update conflict is reported. Furthermore, two delete/update conflicts have occurred, because Harry deleted the class C3 in which Sally added the attributes perimeter and radius. Finally, the violation “Inheritance Cycle” is reported between the classes C2 and C6 which is raised by a constraint in the metamodel.

3 A Holistic Conflict Model

In this section, we present a conceptual representation of changes and conflicts between two independent modifications of a common base model. An outright representation is mandatory for modeling environments to construct a supportive visualization of all occurred conflicts.
3.1 Prerequisites

Before elaborating on the representation of conflicts, we shortly discuss preceding steps which are mandatory to finally explicate conflicts. These steps are (i) capturing the changes performed between two versions of a model, and subsequently, (ii) identifying conflicting pairs of changes.

(i) Capturing changes. A prerequisite for detecting conflicts is to capture the actual changes that have been concurrently performed by two modelers on the same original model. Generally, there are two different techniques to accomplish this task [6]: State-based approaches compare different states, i.e., versions of a model to derive the differences between these states. In contrast, change-based approaches capture changes by observing and recording the modifications while the user performs them.

Recently, it has been widely recognized that the additional knowledge on applied composite operations like refactorings is highly beneficial for versioning. Respecting refactorings enables a faster and better understanding of the modeler’s original intention and enables a smarter conflict detection and resolution [1, 7]. Some change-based approaches allow to directly capture applied refactorings at execution time (e.g., [9]). However, such approaches strongly depend on the modeling environment and only predefined refactorings for a specific modeling language may be detectable. Manually performed refactorings remain unrevealed. To overcome these shortcomings, refactoring occurrences may also be retrospectively detected using state-based approaches as realized in the model versioning system AMOR [1]. Nevertheless, state-based refactoring detection may accomplish a lower precision compared to change-based refactoring detection.

(ii) Detecting conflicts. Having captured all performed changes, conflicts may be detected. In this paper, we consider three kinds of conflicts which are in-depth discussed in [2]. The simplest kind of conflict arises if two opposite changes modify the same feature of a model element in a contradicting way resulting in update/update and delete/update conflicts. Regarding refactorings, another kind of conflict may occur, if the execution of a refactoring is not possible anymore after incorporating the opposite modeler’s changes. Such conflicts are referred to as operation contract violation since opposite changes violate the preconditions of a refactoring. Finally, so called post merge violations may also arise if the merged model violates metamodel constraints.

The first kind of conflict is supported by several approaches like [1, 4, 10, 14, 16]). Additionally, refactorings are regarded in the approaches presented in [10], [16], and [1], however, only the approach introduced in [1] has explicitly specified preconditions of refactorings and, thus, also supports more complex operation contract violations. Violations of the metamodel are usually revealed by reusing existing validation frameworks as done by [1] and [10].

3.2 A Model for Conflicts

The essence of a conflict are the involved model elements, the performed changes as well as the violated constraints. These constraints are either preconditions of a
change or conformance rules defined in the metamodel of a modeling language. Hence, in our conflict model depicted in Fig. 2, we assemble two sources of information for obtaining a conflict report, namely the change report, comprising all applied changes, and additional language specifications formulating language specific operations like refactorings and conformance rules. With this conflict model we may profoundly express the three different kinds of conflicts.

– **ContradictingChange.** A conflict caused by contradicting changes always references two changes which interfere each other. These changes may either be atomic or composite changes. For example, in Fig. 1 the two concurrent updates of C1’s name are contradicting changes as well as the introduction of an attribute into a class which is concurrently deleted.

– **OperationContractViolation.** A conflict due to the violation of an operation contract always involves at least one composite change like a refactoring. This change cannot be performed because another change violates a precondition. Composite changes may be specified with a tool like the Operation Recorder as proposed in [3]. We distinguish between two cases: a composite change is either not applicable because a model element violating the change’s precondition has been added (e.g., class C5 in Fig. 1) or an existing model element necessary for the execution has been changed or deleted (e.g., an attribute of class C4 in Fig. 1).

– **PostMergeViolation.** Furthermore, conflicts may arise if the merged model violates metamodel constraints. For example in Fig. 1, a naively merged version would contain a cyclic inheritance relationship between the classes C2 and C6.

The idea of representing conflicts in terms of a model is not new. Cicchetti et al. [5] recently proposed a metamodel to describe conflict patterns used to
match against a change report for detecting conflicts. In contrast to Cicchetti et al., our approach is designed for the automatic calculation of conflicts by using additional language information like composite change specifications and metamodel constraints. The detection of contradicting changes does not require any additional information. Thus, we are able to derive conflict descriptions automatically.

4 Representing and Visualizing Conflicts in UML Models

In the previous section, we have introduced a model for representing conflicts. However, when it comes to showing the conflicts to the user, appropriate visualization techniques are a must. Thus, in this section we leverage the dynamic extension power of UML profiles by introducing a dedicated change profile and conflict profile (cf. Fig. 4). Both profiles are used to visualize the evolution of a model and occurred conflicts. The design rationale for choosing UML profiles is based on the following requirements:

- **User-friendly visualization**: Merge conflicts as well as the information on performed changes shall be presented in the concrete syntax of UML.
- **Integrated view**: All information shall be visualized within a single diagram to provide a complete overview of conflicts.
- **UML-conform models**: The models incorporating the conflict information shall be compliant with the UML metamodel.
- **No editor modifications**: The visualization of conflicts in UML models shall be possible without modifying the graphical editors of UML tools.
- **Model-based representation**: If models are exchanged between UML tools, the conflict information shall not be lost. Thus, conflicts should be explicitly represented as model elements. Then, conflicts may be resolved later.
UML profiles typically comprise stereotypes, tagged values, and additional constraints stating how profiled UML models may be built. Stereotypes are used to introduce additional modeling concepts to extend standard UML metaclasses. Once a stereotype is specified for a metaclass, the stereotype may be applied to instances of the extended metaclass to provide further semantics. With tagged values, additional properties may be defined for stereotypes. These tagged values may then be set on the modeling level for applied stereotypes. Furthermore, syntactic sugar in terms of icons for defined stereotypes may be configured to improve the visualization of profiled UML models. The major benefit of UML profiles is that profiled models are still compliant to UML and, thus, are naturally handled by current UML tools.

In the remainder of this section, we first present the Change Profile and the Conflict Profile. Second, we elaborate on the algorithm for computing the Conflict Diagram View, i.e., the merged model including change information as well as conflicts in terms of stereotypes and tagged values. Finally, we discuss possible interaction techniques with this conflict diagram view from a user perspective.

4.1 A UML Profile for Conflicts

As depicted in Fig. 2, the conflict model assembles the change report comprising all atomic and composite changes as well as the conflict report which subsumes all detected conflicts. A conflict links to the actual conflicting changes in the change report. This separation is also considered in the UML profile by providing a dedicated profile for visualizing changes and a dedicated profile for visualizing conflicts (cf. Fig. 4). Both profiles are derived from the previously presented conflict model. Please note that the UML profile comprises additional information, e.g., subtypes and properties, which is implicitly stated in the conflict model.

Change Profile. The change profile provides stereotypes for each kind of change. The stereotypes for atomic changes, like adds, updates, and deletions may be applied to all concrete UML concepts. Thus, the stereotype ≪Atomic-Change≫ extends the root metaclass Element of the UML metamodel. In contrast to atomic changes, composite changes involve several model elements. Therefore, we decided to explicitly introduce a UML Collaboration annotated with a ≪CompositeChange≫ stereotype for each composite change. The collaboration links via UML Connectors to the involved model elements to which appropriate ≪Add≫, ≪Delete≫, and ≪Update≫ stereotypes are applied. Finally, for each change the responsible user is saved as meta information.

Conflict Profile. For each of the aforementioned conflict kinds, the conflict profile provides a stereotype with appropriate tagged values. For contradicting changes, the profile provides an ≪Update/Update≫ and ≪Delete/Update≫ stereotype. Both may be applied to any UML element. In contrast to contradicting changes, violations may involve several model elements. Hence, similar as for composite changes, for each violation a UML Collaboration annotated with the respective stereotype ≪OperationContractViolation≫ or ≪PostMergeViolation≫ is introduced. A collaboration refers to elements involved in the violation using UML Connectors. In case of operation contract violations, to add more semantics
to these connectors, they are annotated with respective stereotypes (inspired from graph transformation theory [11]) for marking how the contract is violated by the model element (cf. lower right hand side of Fig. 4).

4.2 Generating the Conflict Diagram View

Resolving conflicts by manually exploring the base model as well as changed models in combination with the change report and a list of conflicts seems to be too cumbersome and error-prone in practice. Thus, we generate a dedicated Conflict Diagram View showing the merged model comprising all relevant changes and detected conflicts at a single glance (cf. Fig. 4 for the running example). This view is obtained as follows:

1. All non-conflicting atomic updates and additions are applied to the common base model. Deletions are skipped, to allow annotating deleted elements with the respective stereotype (e.g., area in class Line in Fig. 4). Also composite changes are left out in this step since they are handled in Step 4.
2. To each changed element, the corresponding change type is annotated by applying the respective stereotype of the change profile (e.g., Point in Fig. 4).
3. Contradicting changes are annotated by applying ≪Delete/Update≫ and ≪Update/Update≫ stereotypes to the involved elements (e.g., Element in Fig. 4). Updated features and its changed values are stored in tagged values.
4. The applied composite changes are considered by checking their preconditions with the merged model. If the preconditions are still valid, they are re-executed on the merged model. If the preconditions are invalid, an operation contract violation is at hand. Since such conflicts involve several model elements, we add a UML Collaboration for each of these conflicts. The added collaboration references (i) model elements to which the composite change has been originally applied (e.g., gray lines from Pull Up Field in Fig. 4), (ii) the elements which are no longer fulfilling the precondition, i.e., all classes must have the field to be pulled up, due to changes by another user (e.g., red line annotated with ≪Delete/Use≫ in Fig. 4), and (iii) elements added by another user which violate the change’s preconditions (e.g., red line annotated with ≪Add/Forbid≫ in Fig. 4).
5. Finally, all post merge violations are marked adding collaborations referring to the involved model elements (e.g., Inheritance Cycle in Fig. 4).

4.3 Interaction with the Conflict Diagram View

The Conflict Diagram View provides several benefits concerning the resolution of the conflicts. First of all, necessary information to resolve the occurred conflicts is provided at a single glance. Furthermore, different diagram filters based on the stereotypes may be used. With the help of these filters, specific kinds of stereotypes, i.e., conflicts, may be hidden enabling the user to focus on a specific conflict scenario. For example, a conflict resolution process can be supported such as firstly representing contradicting changes, subsequently, operation
contract violations, and finally, post merge conflicts. Based on the stereotypes, additional mechanisms for visualizing conflicts are supported by state-of-the-art UML modeling tools. As depicted in Fig. 4, specialized colors are used for stereotyped elements. Moreover, for each stereotype, possible resolution methods may be provided and, in addition, after resolving a conflict the selected resolution is stored for preserving the history of the resolution process. Finally, conflicts may be temporarily tolerated and kept in the model to be handed over to another user as issue report.

5 Conclusions and Future Work

In this paper, we proposed a holistic conflict model for optimistic model versioning. By using information of the modeling language such as metamodel constraints and specifications of refactorings, we are able to automatically detect conflicts which go beyond trivial update/update and delete/update conflicts. For representing and visualizing conflict reports in UML modeling tools, we proposed a UML conflict profile. By this, we achieved a tool independent representation and visualization without any additional implementation effort for editor extensions. We realized this approach within Enterprise Architect\(^6\) which provides powerful visualization and filtering techniques based on UML profiles. The conflict profile and example models are available on our project homepage\(^7\).

In future work, we will perform user studies in order to explore the usability of conflict profiled models. With this, we want to empirically measure how our visualization approach influences the conflict resolution process. Furthermore, we will increase the usage of smart filtering techniques to improve the support

---

\(^6\) [http://www.sparxsystems.eu](http://www.sparxsystems.eu)

\(^7\) [http://www.modelversioning.org/conflict-profile](http://www.modelversioning.org/conflict-profile)
of huge and intensely modified models. We also plan to apply our approach to other Ecore-based modeling languages.

References

Automated Planning for Resolving Model Inconsistencies – A Scalability Study

Jorge Pinna Puissant¹, Ragnhild Van Der Straeten¹,², and Tom Mens¹

¹ University of Mons, 20 Place du Parc, 7000 Mons, Belgium
{jorge.pinnapuissant, tom.mens}@umons.ac.be
² Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium
rvdstrae@vub.ac.be

Abstract. Various approaches have been explored to detect and resolve software model inconsistencies. In this article, we outline our research that uses the artificial intelligence technique of automated planning for the purpose of resolving software model inconsistencies. We discuss the feasibility and scalability of a progression planner, and provide initial results of using a regression planner to improve the scalability.

1 Introduction

In model-driven software engineering (MDE), model inconsistencies are inevitable, because a (software) system description is composed of a wide variety of diverse models, some of which are developed and maintained in parallel, and most of which are subject to continuous evolution. Our research focuses on the activity of model inconsistency resolution. This activity is divided into four steps: (1) Select the inconsistencies that need to be resolved; (2) Identify possible resolution plans to resolve the selected inconsistencies; (3) Perform a cost-benefit analysis of the implementation of each of these resolution plans; (4) Select and apply resolution actions, based on the previous choices [1]. We focus on how to automate step (2) of identifying possible resolution plans.

We propose to use the Automated Planning technique from the Artificial Intelligence domain. To assess whether the proposed technique scales up to large industrial-size software models, we carry out a scalability study using an “off-the-shelf” progression planning tool, and report on initial experiments with a custom-built regression planning tool that we developed in Prolog. We also discuss alternatives to automated planning that may be more appropriate.

2 Automated Planning

In state-of-the-art approaches on inconsistency resolution, resolution rules or resolution generators need to be implemented manually and only one inconsistency at a time is considered [2–7]. Our aim is to tackle the problem of inconsistency resolution by generating possible resolution plans without the need of manually writing resolution rules or writing any procedures that generate choices. The
approach needs to generate valid models with respect to the modelling language and needs to enable the resolution of multiple inconsistencies at once and to perform the resolution in a reasonable time. In addition, the approach needs to be generic, i.e., it needs to be easy to apply it to different modelling languages. We explore Automated Planning, a technique coming from artificial intelligence, for this purpose [8].

Automated planning aims to create plans, i.e., sequences of primitive actions that lead from an initial state to a state meeting a specific predefined goal. To accomplish this, the planner decomposes the world into logical conditions and represents a state as a conjunction of literals. As input the planner needs a planning environment, composed of an initial state, a desired goal and a set of primitive actions that can be performed. The initial state represents the current state of the world. The goal is a partially specified state that describes the world that we would like to obtain. The actions express how each element of a state can be changed. The actions are composed of a precondition and an effect. The effect of an action is executed if and only if the precondition is satisfied. In general a planning approach consists of a representation language used to describe the problem and an algorithm representing the mechanism to solve the problem.

One way to solve planning problems consists in translating them into a satisfiability problem and using a model checker [9]. A more direct approach consists in generating a search space and looking for a solution in this space. Depending on how the state space is traversed, we can distinguish between progression planning and regression planning. Progression planning performs a forward search that starts in the initial state and tries to find a sequence of actions that reaches a goal state. Regression planning starts in the goal state and searches backwards to find a sequence of actions that reach the initial state. In this article, we will compare both approaches for the purpose of finding a model inconsistency resolution plan.

3 Planning for Inconsistency Resolution

Design models can be of different types (e.g. UML, Petri nets, feature models, business process models). They can suffer from many kinds of inconsistencies, such as structural and behavioural inconsistencies. As an example, Figure 1 illustrates a simple class diagram containing two structural inconsistency occurrences of type “Inherited Cyclic Composition” (ICC) and two occurrences of type “Cyclic Inheritance” (CI) [10]. An ICC inconsistency occurs when a composition relationship and an inheritance chain form a cycle that would produce an infinite containment of objects upon instantiation. A first occurrence $ICC_1$ of this type appears in the inheritance chain $\text{Vehicle} \leftarrow \text{Boat} \leftarrow \text{Amphibious Vehicle}$. The second inconsistency $ICC_2$ occurs in the inheritance chain $\text{Vehicle} \leftarrow \text{Car} \leftarrow \text{Amphibious Vehicle}$. A CI inconsistency arises when an inheritance chain forms a cycle. A first occurrence $CI_1$ can be observed in the inheritance cycle involving the classes $\text{Vehicle}$, $\text{Boat}$ and $\text{Amphibious Vehicle}$. The second occurrence $CI_2$ occurs
in the inheritance cycle involving the classes Vehicle, Car and Amphibious Vehicle.

All four inconsistency occurrences share two of the three classes that compose their respective inheritance chains: Vehicle and Amphibious Vehicle. Due to this overlap, the same resolution action can resolve more than one inconsistency occurrence. For example, removing the composition relationship between Vehicle and Amphibious Vehicle solves the two inconsistency occurrences ICC$_1$ and ICC$_2$. Removing the inheritance relationship between Boat and Amphibious Vehicle solves the two inconsistency occurrences ICC$_1$ and CI$_1$. This clearly illustrates that, in order to resolve model inconsistencies in an optimal way, it is important to consider all inconsistencies simultaneously.

Using the example of Figure 1, we illustrate how to create a sequence of inconsistency resolution actions with automated planning. We require as input an initial state (the inconsistent model), a set of possible actions (that change the model) and a desired goal (negation of inconsistencies, see further). Planning requires logic conditions as input, so the whole model environment (e.g. model, meta-model, detection rules) is translated into a conjunction of logic literals.

In 1971, Fikes et al [11] developed a formal planning representation language called STRIPS. In 1989, Pednault [12] developed a more advanced and expressive language called ADL. It applies the open world principle and allows to use negative literals and disjunction. PDDL [13] is a generic language (Planning Do- main Definition Language) allowing to represent the syntax of STRIPS, ADL and other languages. We will use the Lisp-like syntax of PDDL [14] to explain our approach. Each logic literal is a tuple represented between parentheses. The tuple starts with the name of the literal, followed by pairs of variable names and their type (separated by a “-“). There are no primitive types in PDDL.

The metamodel for class diagrams is translated to PDDL as follows. Each metamodel element can be referred to by a unique id.

(Class ?id - class_id ?name - String)
(Generalisation ?id - g_id ?label - String ?child_class - class_id ?parent_class - class_id)
(Association ?id - a_id ?name - String ?ass_end_1 - ae_id ?ass_end_2 - ae_id)

The initial state is expressed as a conjunction of literals, and represents the current world. In our case the initial state will be the inconsistent model. The
initial state can be represented either by using the complete model, or by using a partial model that contains only those elements that are involved in one or more inconsistency occurrences. Below is an example of a partial model (conforming to the aforementioned metamodel), containing only the elements that are involved in the inconsistency occurrences, shown in the shaded part of Figure 1.

(Class c1 Vehicle)
(Class c5 Boat)
(Class c6 Car)
(Class c9 Amphibious_Vehicle)
(Generalisation g4 label4 c5 c1)
(Generalisation g5 label5 c6 c1)
(Generalisation g8 label8 c9 c5)
(Generalisation g9 label9 c9 c6)
(Generalisation g10 label10 c1 c9)
(Association_End ae1 c9 role1 star one non)
(Association_End ae2 c1 role2 one one yes)
(Association a1 ass1 ae1 ae2)

The set of actions that can be performed to change a model are represented in terms of a precondition that must hold before the execution and the action to execute. The set of actions corresponds to the elementary operations (create, modify and delete) of the different types of model elements that can be derived from the metamodel. Combined with the logic literals of the metamodel, this allows us to compute the list of all possible actions. As an example, the specification of modify Association_Name is given below.

(:action modify_Association_Name
 :parameters (?id - id ?name - String ?ass_end_1 - ae_id
 ?ass_end_2 - ae_id ?new_name - String)
 :precondition (Association ?id ?name ?ass_end_1 ?ass_end_2)
 :effect (when (not (= ?name ?new_name))
 (and (not (Association ?id ?name ?ass_end_1 ?ass_end_2))
 (Association ?id ?new_name ?ass_end_1 ?ass_end_2)))
)

The desired goal is a partially specified state, represented as a conjunction of literals using logic quantification. It specifies the objective we want to reach, namely the absence of model inconsistencies. To achieve this we can use two alternatives: (1) the negation of the inconsistency occurrences; or (2) the negation of the inconsistency detection rules. An inconsistency detection rule is a conjunction of literals representing a pattern that, if matched in the model, detects inconsistency occurrences. Below we give an example of the “Inherited Cyclic Composition” detection rule. It only specifies an inheritance chain involving three classes because PDDL syntax does not allow to express transitive closure to make the rule more generic.

(exists (?a - class_id ?b - class_id ?c - class_id)
 (and
 (exists (?g - g_id ?Label - g_label)
 (Generalisation ?g ?Label ?c ?a))
 (exists (?g - g_id ?Label - g_label)
 (Generalisation ?g ?Label ?b ?a))
 (exists (?ae - ae_id ?role - ae_role ?upper - upper_cardinal ?lower - lower_cardinal)
 (Association_End ?ae a ?role ?upper ?lower yes))
 (exists (?ae - ae_id ?role - ae_role ?upper - upper_cardinal ?composite - boolean)
 (Association_End ?ae a ?role ?upper one ?composite))
))
Alternative (1) above will only be able to resolve inconsistency occurrences that have already been identified previously. Alternative (2) has the advantage that it can be used to detect and resolve inconsistency occurrences at the same time, but suffers from scalability problems (see further). In both alternatives logic negation is used to express the absence of inconsistencies in the resulting model. This implies that we need a planning language that allows the use of disjunction and negative literals in the goal.

A plan is a sequence of actions that transforms the initial model into a model that satisfies the desired goal (i.e., a consistent model). A plan is generated automatically by the planning algorithm, without relying on any domain-specific information. Moreover, the generated resolution plan does not lead to ill-formed models (that do not conform to their metamodel) as long as all metamodel constraints are given as part of the problem specification. A complete resolution plan, containing only two actions, that solves the four inconsistency occurrences of the motivating example is given below:

```plaintext
delete_Generalisation: (Generalisation g10 label10 c1 c9)
modify_Association_End_Lower_Multiplicity
   from: (Association_End ae1 c9 role1 star one non)
   to: (Association_End ae1 c9 role1 star zero non)}
```

4 The Experiment

Our scalability study aims to assess whether planning algorithms relying on state space search are sufficiently expressive and scalable to be used for resolving inconsistencies in software models. To assess this, first we experimented with a state-of-the-art progression planner. Because it does not scale well (as will be shown below), we started to build our own regression planner. To evaluate both planners, we have carried out several experiments, using a 64-bit Apple MacBook with 2.4 GHz Intel Core 2 Duo processor and 4GB RAM, 2.9GB of which were available for the experiments. In order to remove noise, each experiment was executed 10 times and the average time and standard deviation was computed. For both considered planners, the generated resolution plans were always complete (i.e., they removed all occurrences of all inconsistencies that were taken into consideration). Typically, there are many ways in which inconsistencies can be resolved. Since both considered planners look for the shortest path in the search space, they always provide a resolution strategy that is minimal in the number of actions required to resolve all inconsistencies.

Progression planner. To choose a progression planner, we surveyed the state-of-the-art on existing planner tools. An important constraint was that the planning language needs to support disjunction and negative literals (see section 3). FF (for “Fast-Forward Planning System” [15]) is a heuristic state-space progression planner, and the only one we found to be able to properly deal with negation. To be precise, FF supports the PDDL language with full ADL subset support, including positive and negative literals, conjunction and disjunction, negation, typing, and logic quantification in the desired goal.
Table 1: Timing results for experiments with FF on case study of section 3

<table>
<thead>
<tr>
<th>Experiment number</th>
<th>Initial state</th>
<th>Desired Goal: Negation of</th>
<th>Average time (in seconds)</th>
<th>Std deviation (in milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>complete model</td>
<td>inconsistency rules</td>
<td>out of memory</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>partial model</td>
<td>inconsistency rules</td>
<td>out of memory</td>
<td>N/A</td>
</tr>
<tr>
<td>3</td>
<td>complete model</td>
<td>inconsistency occurrences</td>
<td>14.84</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>partial model</td>
<td>inconsistency occurrences</td>
<td>0.268</td>
<td>4</td>
</tr>
</tbody>
</table>

Our first series of experiments aimed to explore the impact of different ways to give the input for the planner, as explained in section 3. The initial state can be specified by giving a complete model or a partial model containing only those elements that are involved in the inconsistency occurrences (shaded part of Figure 1). The desired goal can either contain a negation of the inconsistency detection rules or a negation of the inconsistency occurrences. Table 1 presents the timing results for each combination of choices using FF on the model of Figure 1. Because Table 1 clearly shows that using the negation of inconsistency rules for the desired goal gives rise to an out of memory, the remaining experiments only use the negation of inconsistency occurrences as desired goal.

Our second series of experiments aims to assess how the generation of a minimal resolution plan scales up as the models increase in size. To illustrate the advantage of using partial models as opposed to complete models as initial state, we reran experiment 3 of Table 1, while artificially augmenting the size of the model by gradually adding a number of isolated classes. For 20 added isolated classes the progression planner took more than 5 hours. A regression analysis reveals an exponential relation with a very good fit of $R^2 = 0.982$ for the progression planner.

We also reran experiment 4 of Table 1 for models of increasing size. As the introduction of isolated classes does not affect the partial model used as initial state, the timing results remain constant, irrespective of how many isolated classes are added. To assess the effect of an increase of the size of the partial model on the time needed to compute a resolution plan, we artificially augmented the size of the model by gradually increasing the length of the inheritance chains involved in the inconsistency occurrences of Figure 1. Figure 2a shows the timing results obtained with FF, after adding between 1 and 8 intermediate superclasses. The increase in time for computing a minimal resolution plan appears to follow an exponential growth (with $R^2 = 0.995$).

We verified whether the number of inconsistency occurrences to be resolved affected the timing results. To achieve this, we restricted the desired goal to generate resolution plans that resolve only 2 or 3 inconsistency occurrences in the partial model of Figure 1. Since this did not change the size of the partial

---

3 The second best regression was a quadratic polynomial with a very good fit of 0.977.
model a lot, it did not have any significant impact on the performance, as the growth rate and timing results were still similar to what we found in Figure 2a.

As a final experiment, we added a new type of metamodel element to represent class attributes in a model, and we extended the set of actions with three new actions to create, modify and delete attributes in a model. As long as the initial state and desired goal do not use these attributes, the timing results do not change. However, when we start to artificially augment the initial model by gradually adding extra attributes to the existing classes of the model, without modifying the desired goal, the time needed to generate a minimal resolution plan starts to increase. Figure 2b shows the timing results obtained, when adding 1 to 20 attributes to classes in the model. The growth appears to follow a quadratic polynomial (with $R^2 = 0.994$).

**Regression planner.** We did not find a readily available regression planner that fit our needs. Therefore, based on the algorithms explained in [16] we started implementing our own regression planner in Prolog. The way in which to represent metamodels, models, actions and inconsistencies is similar to what we explained in section 3. In addition, the logic programming language Prolog provides more expressiveness than FF. For example, we can express transitive closure in a straightforward way.

Experiments with our custom-built regression planner are underway. We have already repeated experiment 3 of Table 1. The results, shown in Figure 3, show that the regression planner outperforms the progression planner with several orders of magnitude. In addition, the curve for the regression planner increases more slowly. For 20 added isolated classes, the regression planner takes 1.21 seconds. A regression analysis reveals a quadratic polynomial with extremely good fit of $R^2 = 0.999$ for the regression planner.\(^4\)

---

\(^4\) Two other regression models we verified had a lower goodness of fit: 0.982 for an exponential model and 0.763 for a power curve.

\(^5\) The second best regression was an exponential with a very good fit of 0.990.
Fig. 3: Timing results for progression planning (blue circles) and regression planning (red triangles) using a complete model. The y-axis represents the time in seconds on a logarithmic scale. The x-axis represents the number of isolated classes added to the initial model.

5 Discussion and Future Work

Let’s discuss a number of points that require more thorough investigation. Using inconsistency rules (as opposed to inconsistency occurrences) in the desired goal leads to out of memory problems, as the search space becomes too big. Nevertheless, a distinct advantage of inconsistency rules is that it enables detection and resolution of inconsistencies at the same time. An approach based on inconsistency occurrences requires a preliminary phase in which the inconsistency occurrences have been detected.

Providing the complete model as initial state may not be realistic, as it gives out of memory errors because the search space becomes too big. Using a partial model as input resolves this problem (at least in the experiments we carried out).

The type of planner used significantly affects the results. We compared the progression planner FF with our own regression planner. While FF does the job, it suffers from scalability and has very poor timing results. FF appears to be optimized for a range of problems in which the search tree is typically narrow and deep, and in which negation is seldomly needed. For the purpose of model inconsistencies, we require negation in the desired goal, and deal with search trees that are wide (there are many actions to consider at each step) and shallow (the total number of actions in the resolution plan is roughly proportional to the number of inconsistency occurrences that need to be resolved). Regression planners appear to perform significantly better for this type of problem because they restrict the search space by excluding many irrelevant actions. Moreover, since we implemented a regression planner ourselves in Prolog, we can still optimise it further to take into account specificities about the problem domain.

Our experiments were based on a single case study (a small class diagram with 4 inconsistency occurrences of two different types). The results of our experiments may therefore be biased. A proper validation would require a wide range of models, of varying sizes and using different metamodels. Note that the automated planning approach does not depend on a particular metamodel, so it is easy to apply it to the resolution of different types of models.
From a usability point of view, it is fairly straightforward to write a convertor that automatically transforms models (and metamodels) into the logic format required as input by the planner, and to output the resolution plan in the form of a model transformation that is able to correct the inconsistent model. This would avoid users of the approach to learn a new language.

Automated planners compute a resolution plan that is minimal in the number of actions to carry out to resolve all inconsistency occurrences. This does not mean that the resolution plan is also unique. More generally speaking, a minimal plan may not always be the most appropriate solution. In order to investigate what is the most appropriate resolution plan, one could generate all possible resolution plans (up to a certain size), and let the designer choose the most appropriate one (based on some criteria to be defined).

Most automated planners suffer from lack of expressiveness. For example, FF was unable to express transitive closure, primitive types and numbers. Our regression planner in Prolog does not have this problem. Of course, one always needs to find a trade-off between more expressiveness and better performance. The more expressive, the wider range of inconsistencies that can be detected and resolved, but the longer it may take to find a resolution plan (if at all).

Harman [17] advocates the use of search-based approaches in software engineering. This includes a wide variety of different techniques and approaches such as metaheuristics, local search algorithms, automated learning, genetic algorithms. We believe that these techniques could be applied to the problem of model inconsistency management, as it satisfies at least three important properties that motivate the need for search-based software engineering: the presence of a large search space, the need for algorithms with a low computational complexity, and the absence of known optimal solutions.

6 Conclusion

We explored the use of automated planning, a logic-based approach originating from artificial intelligence, for the purpose of resolving model inconsistencies. We are not aware of any other work having used this technique for this particular purpose. We applied a progression planner on a simple case study, and carried out experiments to assess the scalability. Our results reveal that the approach is feasible but suffers from various scalability problems. Nevertheless, there is still room for improvement. Initial results with a regression planner yield a significant improvement but further experiments are necessary to confirm this. Alternatively, other search-based approaches could be explored to deal with model inconsistency resolution.

Acknowledgements. This work has been partially supported by (i) the F.R.S. – FNRS through FRFC project 2.4515.09 “Research Center on Software Adaptability”; (ii) research project AUWB-08/12-UMH “Model-Driven Software Evolution”, an Action de Recherche Concertée financed by the Ministère de la Communauté française - Direction générale de l’Enseignement non obligatoire et de la Recherche scientifique, Belgium; (iii) Avec le soutien de Wallonie - Bruxelles International et du Fonds de la Recherche Scientifique, du Ministère Français des Affaires étrangères et européennes,
du Ministère de l’Enseignement supérieur et de la Recherche dans le cadre des Partenariats Hubert Curien.

References

A Manifesto for Semantic Model Differencing

Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe

Dept. of Computer Science 3, Software Engineering
RWTH Aachen University, Germany
http://www.se-rwth.de/

Abstract. Models are heavily used in software engineering and together with their systems they evolve over time. Thus, managing their changes is an important challenge for system maintainability. Existing approaches to model differencing concentrate on heuristics matching between model elements and on finding and presenting differences at a concrete or abstract syntactic level. While showing some success, these approaches are inherently limited to comparing syntactic structures.

This paper is a manifesto for research on semantic model differencing. We present our vision to develop semantic diff operators for model comparisons: 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. In particular, if the models are syntactically different but there are no diff witnesses, the models are semantically equivalent. We demonstrate our vision using two concrete diff operators, for class diagrams and for activity diagrams. We motivate the use of semantic diff operators, briefly discuss the algorithms to compute them, list related challenges, and show their application and potential use as new fundamental building blocks for change management in model-driven engineering.

1 Introduction

Effective change management, a major challenge in software engineering in general and in model-driven engineering in particular, has attracted much research efforts in recent years (see, e.g., [7, 8, 13–15]). Due to iterative development methodologies, changing requirements, and bug fixes, models continuously evolve during the design, development, and maintenance phases of a system’s lifecycle. Managing their changes using formal methods to follow their different versions over time is thus an important task. Fundamental building blocks for this task are diff operators one can use for model comparisons.

Existing approaches to model differencing concentrate on matching between model elements using different heuristics related to their names and structure and on finding and presenting differences at a concrete or abstract syntactic level. While showing some success, these approaches are also limited. Models that are syntactically very similar may induce very different semantics (in the sense of ‘meaning’ [10]), and vice versa, models that semantically describe the same system may have rather different syntactic representations. Thus, a list
of syntactic differences, although accurate, correct, and complete, may not be able to reveal the real implications these differences have on the correctness and potential use of the models involved. In other words, such a list, although easy to follow, understand, and manipulate (e.g., for merging), may not be able to expose and represent the semantic differences between two versions of a model, in terms of the bugs that were fixed or the features (or new bugs . . .) that were added.

This paper is a manifesto for research on semantic model differencing. We present our vision to develop semantic diff operators for model comparisons: operators whose input consists of two models and whose output is a set of diff witnesses, instances of the first model that are not instances of the second. Such diff witnesses serve as concrete proofs for the real change between one version and another and its effect on the meaning of the models involved.

We demonstrate our ideas using two examples of concrete semantic diff operators, for class diagrams (CDs) and for activity diagrams (ADs), called cddiff and addiff, respectively. Given two CDs, cddiff outputs a set of diff witnesses, each of which is an object model that is an instance of the first CD and not an instance of the second. Given two ADs, addiff outputs a set of diff witnesses, each of which is a finite action trace that is possible in the first AD and is not possible in the second. Each operator considers the specific semantics of the relevant modeling languages, e.g., in terms of multiplicities, inheritance, etc. for CDs, and decision nodes, fork nodes, etc. for ADs.

In addition to finding concrete diff witnesses (if any exist), our operators can be used to compare two models and decide whether one model semantics includes the other model semantics (the latter is a refinement of the former), whether they are semantically equivalent, or whether they are semantically incomparable (each allows instances that are not allowed by the other). When applied to the version history of a certain model, such an analysis provides a semantic insight into the model’s evolution, which is not available in existing syntactic approaches.

We have already implemented prototype versions of cddiff and addiff: all examples shown in this paper were computed by our prototype implementations. Section 4 gives a brief overview of the algorithms and tools we have used.

It is important not to confuse diffing with merging. Merging is a very important problem, dealing with reconciling the differences between two models that have evolved independently from a single source model, by different developers, and now need to be merged back into a single model (see, e.g., [4, 8, 13]). Diffing, however, is the problem of identifying the differences between two versions, for example, an old version and a new one, in order to better understand the course of a model evolution during some step of its development. Thus, diff witnesses are not conflicts that need to be reconciled. Rather, they are proofs of features that were added or bugs that have been fixed from one version to another along the history of the development process.

Finally, our vision of semantic diffing does not come to replace existing syntactic diffing approaches. Rather, it is aimed at augmenting and complementing existing approaches with capabilities that were not available before. As seman-
tic differencing is so different from existing syntactic differencing approaches, it brings about new research challenges. We overview these challenges in Section 5.

The next section presents motivating examples, demonstrating the unique features of our vision. Section 3 presents a formal definition of a generic semantic diff operator and its specializations for CDs and ADs. Section 4 briefly describes the algorithms used to compute the two operators and their prototypes implementations, and Section 5 discusses new challenges emerging from our vision. Related work is discussed in Section 6 and Section 7 concludes.

2 Examples

We start off with a number of motivating examples, demonstrating the unique features of our vision.

Example 1 Consider \(cd_{1}.v_{1}\) 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 two bugs in this model: first, employees should not be assigned more than two tasks, and second, managers are also employees, and they can handle tasks too.

Following this design review, the engineers created a new version \(cd_{1}.v_{2}\), shown in the same figure. The two versions share the same set of named elements but they are not identical. Syntactically, the engineers added an inheritance relation between Manager and Employee, and set the multiplicity on the association between Employee and Task to 0..2. What are the semantic consequences of these differences?

Using the operator \(cddiff\) we can answer this question. \(cddiff(cd_{1}.v_{1}, cd_{1}.v_{2})\) outputs \(om_{2}\), shown in Fig. 1, as a diff witness that is in the semantics of \(cd_{1}.v_{1}\) and not in the semantics of \(cd_{1}.v_{2}\); thus, it demonstrates that the bug of having more than two tasks per employee was fixed. In addition, \(cddiff(cd_{1}.v_{2}, cd_{1}.v_{1})\) outputs \(om_{1}\), shown in Fig. 1, as a diff witness that is in the semantics of \(cd_{1}.v_{2}\) and not in the semantics of \(cd_{1}.v_{1}\). Thus, the engineers should perhaps check with the domain expert whether the model should indeed allow managers to manage themselves.

Example 2 \(cd_{5}.v_{1}\) of Fig. 2 is another class diagram from this model of company structure. In the process of model quality improvement, an engineer has suggested to refactor it by introducing an abstract class Person, replacing the association between Employee and Address by an association between Person and Address, and redefining Employee to be a subclass of Person. The resulting suggested CD is \(cd_{5}.v_{2}\).

Using \(cddiff\) we are able to prove that despite the syntactic differences, the semantics of the new version is equivalent to the semantics of the old version, formally written \(cddiff(cd_{5}.v_{1}, cd_{5}.v_{2}) = cddiff(cd_{5}.v_{2}, cd_{5}.v_{1}) = \emptyset\). The refactoring is indeed correct and the new suggested version can be committed.

Example 3 AD \(ad.v_{1}\) of Fig. 3 describes the company’s work flow when hiring a new employee. Roughly, first the employee is registered. Then, if she is an internal employee, she gets a welcome package, she is assigned to a project and added to
the company’s computer system (in two parallel activities), she is interviewed and gets a manager report, and finally her payments are authorized. Otherwise, if the new employee is external, she is only assigned to a project before her payments are authorized.

After some time, the company deployed a new security system and every employee had to receive a key card. A revised work flow was created, as shown in ad.v2 of Fig. 3.

Later, a problem was found: sometimes employees are assigned to a project but cannot enter the building since they do not have a key card yet. This bug was fixed in the next version, ad.v3, shown in Fig. 4. Finally, the company has decided that external employees should report to managers too. Thus, the merge between the two branches for internal and external new employees has moved ‘up’, in between the interview and the report nodes. The resulting 4th version of the work flow, ad.v4, is shown in Fig. 4.
Comparing \textit{ad.v1} and \textit{ad.v2} using \textit{addiff} reveals that they are incomparable: some executions of \textit{ad.v1} are no longer possible in \textit{ad.v2}, and some executions of \textit{ad.v2} were not possible in \textit{ad.v1}. Moreover, it reveals that handling of internal employees has changed, but handling of external ones remained the same between the two versions.

Comparing \textit{ad.v2} and \textit{ad.v3} reveals that the latter is a refinement of the former: \textit{ad.v3} has removed some traces of \textit{ad.v2} and did not add new traces. In particular, \textit{addiff(ad.v2, ad.v3)} shows that the trace where a person is assigned to a project before she gets a security card was possible in \textit{ad.v2} and is no longer possible in \textit{ad.v3}, i.e., it demonstrates that the bug was fixed.

Finally, comparing \textit{ad.v3} and \textit{ad.v4} using \textit{addiff} reveals that although hiring of external employees has changed between the two versions, hiring of internal employees did not: \textit{addiff(ad.v3, ad.v4)} contains a single trace, where the employee is external, not internal. That is despite the syntactic change of moving the merge node from after to before the report node, which is part of the handling of internal employees.

### 3 Formal Definitions

Consider a modeling language $ML = (\text{Syn}, \text{Sem}, \text{sem})$ where $\text{Syn}$ is the set of all syntactically correct (i.e., well-formed) expressions (models) according to some syntax definition, $\text{Sem}$ is a semantic domain, and $\text{sem}: \text{Syn} \rightarrow \mathcal{P}(\text{Sem})$ is a function mapping each expression $e \in \text{Syn}$ to a set of elements from $\text{Sem}$ (see [10]).

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

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

---

**Fig. 3.** Version 1 and version 2 of the hire employee work flow.
Note that \( \text{diff} \) is not symmetric, \( \text{diff}(e_1, e_1) = \emptyset \), and \( \text{diff}(e_1, e_2) \cap \text{diff}(e_2, e_1) = \emptyset \). The elements in \( \text{diff}(e_1, e_2) \) are called \( \text{diff witnesses} \). We define specializations of \( \text{diff} \) for CDs and ADs.

Our semantics of CDs is based on [9] and is given in terms of sets of objects and relationships between these objects. More formally, the semantics is defined using three parts: a precise definition of the syntactic domain, i.e., the syntax of the modeling language CD and its context conditions (we use MontiCore [12, 17] for this); a semantic domain - for us, a subset of the System Model (see [5, 6]) OM, consisting of all finite object models; and a mapping \( \text{sem} : \text{CD} \rightarrow \mathcal{P}(\text{OM}) \), which relates each syntactically well-formed CD to a set of constructs in the semantic domain OM. For a thorough and formal account of the semantics see [6].

To make the operator \( \text{cddiff} \) computable and finite, we bound the number of objects in the witnesses we are looking for. Thus, we define a family of bounded operators. Formally:

**Definition 2 (cddiff).** \( \forall k \geq 0, \text{cddiff}_k(cd_1, cd_2) = \{om | om \in \text{sem}(cd_1) \land om \notin \text{sem}(cd_2) \land |om| \leq k\} \), where \( |om| \) is the maximal number of instances per class in \( om \).

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 from an initial to a final node, considering interleaving execution of fork branches, the guards on decision nodes etc. (a formal and complete semantics of ADs is outside the scope of this paper).

In differencing ADs, we are looking only for shortest witnesses: diff traces that have another diff trace as prefix are not considered interesting. Formally:

*Fig. 4.* Version 3 and version 4 of the hire employee work flow
Definition 3 (addiff). \( \text{addiff}(ad_1, ad_2) = \{ tr | tr \in \text{sem}(ad_1) \land tr \notin \text{sem}(ad_2) \land \not\exists tr' : tr' \in \text{sem}(ad_1) \land tr' \notin \text{sem}(ad_2) \land tr' \sqsubseteq tr \} \).

4 Implementations and Applications

To evaluate our vision and demonstrate its feasibility, we have defined and implemented prototype versions of \( \text{cddiff} \) and \( \text{addiff} \). Indeed, all examples shown in the previous section have been computed by our prototype implementations.

We compute a variant of \( \text{cddiff} \) using a transformation to Alloy [11]. Given two CDs, \( cd_1 \) and \( cd_2 \), we construct a single Alloy model consisting of the joint set of class signatures from the two CDs and a set of predicates that describe the relations between them in each of the CDs. We do not compute all instances of each CD and compare the two sets of instances; rather, we define a diff predicate, which specifies that all the \( cd_1 \) predicates hold and that at least one of the \( cd_2 \) predicates does not hold. We then use the Alloy Analyzer to compute instances of this diff predicate: these instances represent object models of the first CD that are not object models of the second CD. The transformation to Alloy considers the semantics of CDs, including multiplicities, inheritance, singleton and abstract classes etc.

Our implementation of \( \text{cddiff} \) can be used to compute diff witnesses, if any, or to show that no diff witnesses exist (up to a user-defined bound on the number of objects of each class in the model).

We compute a variant of \( \text{addiff} \) by modeling ADs as finite state machines, and defining a transformation to SMV [2]. Given two ADs, \( ad_1 \) and \( ad_2 \), we construct two SMV modules whose possible execution traces are exactly the set of possible traces of each of the ADs. We then use BDD-based algorithms, implemented using JTLV APIs [16], to find whether there are traces of \( ad_1 \) that are impossible in \( ad_2 \). The transformation to SMV and the algorithms used consider the semantics of ADs, including input variables, guarded branching in decision nodes, parallel interleaving execution following fork nodes, etc. The transformation is linear to the size of the ADs, and the BDD-based algorithms are polynomial to the size of the state space of the ADs.

Our implementation of \( \text{addiff} \) can be used to compute diff witnesses, each of which is a finite trace which is a sequence of actions possible in one AD and not possible in the other (a trace includes the values of its input variables). If no such traces are found, we know that all traces of the first are also possible in the second, i.e., that the first is a refinement of the second. If, in addition, no such traces are found when reversing first and second, we know that the two ADs have equal semantics: their syntactic differences, if any, have no effect on their meaning.

We have integrated our implementations into Eclipse plug-ins. The plug-ins allow an engineer to compare two models from a project or two versions of a model from the history of a version repository. The engineer can then browse the diff witnesses that were found, if any.
Moreover, we have used \texttt{addiff} and \texttt{cddiff} to implement a \texttt{COMPARE} command, used to compare two selected models and output one of four answers: $\equiv$ if the two models are semantically equivalent, $\prec$ or $\succ$, if the second (first) is a semantic refinement of the first (second), and $\not\equiv$ if the two are incomparable, that is, if each of them allows instances (i.e., object models, traces) not possible in the other (in the case of \texttt{cddiff} the results of \texttt{COMPARE} are limited by the user-defined bound). \texttt{COMPARE} can be integrated with existing SVN history view, to provide a high-level semantic differencing summary of a model’s evolution.

The details of the above transformations and algorithms for \texttt{cddiff} and \texttt{addiff}, and their related Eclipse plug-ins, are omitted from this workshop paper. We hope to present them in detail in follow-up publications.

5 Challenges

Semantic differencing is rather different from syntactic differencing approaches, so it raises a number of new research challenges.

5.1 Computation

Computing diff witnesses may not be algorithmically easy and sometimes even impossible. When computable, its complexity depends on the specific modeling language semantics at hand. For example, computing \texttt{cddiff} requires the use of a constraint solver (such as Alloy); to make it tractable, it must be bounded (see Section 3). Computing \texttt{addiff} requires a traversal of the state space induced by the ADs at hand. Depending on the use of fork nodes, input variables, and guards, this state space may be exponential in the size of the ADs themselves.

In general, depending on the available syntactic concepts and the semantics of the relevant modeling language, computing diff witnesses may be undecidable. In some cases, the set of computed witnesses may be sound but incomplete: all computed witnesses are indeed correct, but there may be infinitely many others that are harder to find. Thus, for each modeling language, a language specific diff operator needs to be defined and a new algorithm needs to be developed for its computation. Abstraction/refinement methodologies, partial-order reductions, and other approaches may be required in order to improve the efficiency of the computations and allow them to scale.

5.2 Presentation

To be useful, diff witnesses must be presented textually or visually to the engineer. Just like for computation, the presentation of diff witnesses is language specific; it depends on the specific modeling language of the models involved and its semantics. For example, for \texttt{cddiff}, differencing object models may be visually presented using generated object diagrams; for \texttt{addiff}, differencing traces may be visually presented on the ADs themselves, e.g., by coloring and numbering the nodes that participate in the diff trace on both diagrams, from the initial
node up until the point where the two diagrams differ. Alternatively, one may use a collaboration diagram like notation, possibly with the aid of animation.

Moreover, as there may be (possibly infinitely) many diff witnesses, it is necessary to define sorting and filtering mechanisms, to select the 'most interesting' witnesses for presentation and efficiently iterate over them at the user's request.

5.3 Integration with syntactic differencing

Many works have suggested various syntactic approaches to model differencing (see Section 6). It may be useful to combine syntactic differencing with semantic differencing, for example:

- Extend the applicability of semantic diffing in comparing models whose elements have been renamed or moved in the course of evolution, by applying a syntactic matching before running a semantic diffing: this would result in a mapping plus a set of diff witnesses.
- Use information extracted from syntactic diffing as a means to localize and thus improve the performance of semantic diffing computations.

6 Related Work

The challenge of model change management and versioning has attracted much research efforts in recent years. In particular, many works have investigated various kinds of model comparisons. We review some of these briefly below.

[3] describes the difference between two models as a sequence of elementary transformations, such as element creation and deletion and link insertion and removal; when applied to the first model, the sequence of transformations yields the second. A somewhat similar approach is presented in [13] in the context of process models, focusing on identifying dependencies and conflicts between change operations. [8] presents the use of a model merging language to reconcile model differences. Comparison is done by identifying new/old MOF IDs and checking related attributes and references recursively. Results include a set of additions and deletions, highlighted in a Diff/Merge browser. [15] compares UML documents by traversing their abstract-syntax trees, detecting additions, deletions, and shifts of sub-trees.

As the above shows, some works go beyond the concrete textual or visual representation and have defined the comparison at the abstract-syntax level, detecting additions, removals, and shifts operations on model elements. However, to the best of our knowledge, no previous work considers model comparisons at the level of the semantic domain, as suggested in our vision.

Some works, e.g., [1, 18], use similarity-based matching before actual differencing. As our vision focuses on semantics, it assumes a matching is given. Semantic diffing can be applied after the application of matching algorithms.
7 Conclusion

In this paper we described our vision on semantic diff operators for model comparison, as new fundamental building blocks for change management in model-driven engineering. We motivated our vision with examples, and gave a brief overview of the formal background and the algorithms used in our prototype implementations. Finally, we listed new research challenges that emerge from our vision, related to the computation and presentation of semantic model differences.

Acknowledgments We thank Assaf Marron and the anonymous reviewers for their comments on a draft of this paper.

References

Towards Transformation Migration After Metamodel Evolution

David Méndez¹, ², Anne Etien², Alexis Muller², and Rubby Casallas¹

TICSw Research Group, Universidad de los Andes, Colombia
{df.mendez73,rcasalla}@uniandes.edu.co

INRIA Lille Nord Europe, LIFL CNRS UMR 8022, Université Lille 1, France
{Anne.Etien,Alexis.Muller}@inria.fr

Abstract. Metamodels evolve because of several reasons such as design refinement and software requirement changes. When this happens, transformations defined in terms of those metamodels might become inconsistent and migration would be necessary. Due to the lack of methodology support, transformation migration is mostly ad hoc and manually performed. Besides, the growing complexity and size of transformations make this task difficult and error prone. In this paper, we present some preliminary results of our current work in transformation migration.

Keywords: Metamodel Evolution, Model Transformation Adaptation

1 Introduction

Model to model transformations take models conformant to source metamodels and produce models conformant to target metamodels. When source and/or target metamodels evolve, transformations might become inconsistent and updates would be necessary. We extend the vocabulary presented in [5] and call transformation migration the development activity in which transformations are updated after metamodel evolution. Due to the lack of methodology support, transformation migration is mostly ad hoc and manually performed. Besides, the growing complexity and size of transformations make this task difficult and error prone.

Another difficulty of transformation migration is that transformation consistency is not well defined yet. As a consequence, detection of transformation inconsistencies could be a hard task if those inconsistencies are not highlighted as syntax errors in the transformation code. Besides, even if we could detect transformation impact, re-establish consistency requires knowledge and understanding about the transformation and the performed changes.

In this paper we present our preliminary results in transformation migration. We specify transformation consistency by studying the relationship between transformation and metamodels, we called it domain conformance. In the methodological field, we propose a set of tasks that should be performed in order to re-establish consistency after metamodel evolution.
The remainder of this paper is organized as follows. Section 2 explores the related work in evolution of MDE artifacts. Section 3 introduces some preliminary ideas about what is exactly transformation consistency. Section 4 presents the way in which we believe transformation migration should be performed. Section 5 illustrates this approach by using the well-known transformation from class to relational database model and the evolution of the UML class diagram from version 1.5 to version 2.0. Finally, section 7 presents the future work and conclusions.

2 Related Work

Transformations are not the only artifacts impacted by metamodel evolution. In fact, when a metamodel evolves all the artifacts that are defined in terms of it might become inconsistent. Currently, there are several approaches that support this problem, most of them focused on resolving inconsistencies occurring in models after metamodel evolution, i.e. re-establish conformance relationship.

Rose et al. [6] propose a classification of model migration approaches. This classification highlights three ways to identify needed model updates: 1) manually, 2) based on operators, and 3) by using metamodel matching. In manually approaches, like in [4] or [6], updates are defined by hand. In operators based approaches, like [3], the metamodel changes are defined in terms of co-evolutionary operators. Those operators define conjointly the evolution on the metamodel and its repercussions on the models. Finally, in metamodel matching, like [2], versions of metamodels are compared and differences between them are used to semi-automatically infer a transformation that expresses models updates.

The relationship between metamodels and transformations differs from the relationship between metamodels and models. Hence, transformation migration has to be performed differently to model migration. There are some approaches focused on transformation migration, some of them are presented in the remainder of this section. In [8], the authors are aware that metamodel evolution might imply transformation migration. However, they do not propose a transformation migration approach. They define model typing that establishes the metamodel evolution spectrum in which a transformation remains valid. In [7], Roser et al. present an approach that enables semi-automatic transformation migration after metamodel evolution. However, they only support the cases in that the source and target metamodels belong to the same knowledge area represented by a reference ontology. In our work, we aim to propose a solution for transformation migration even if the evolution contains changes that invalidate the transformation and there is not an ontology relating source and target metamodels.

3 Transformation Consistency

A transformation is defined in terms of a set of metamodels. Hence, to define transformation consistency we need to explicitly specify the relationship between
transformations and metamodels. The name we propose for that relationship is  *domain conformance*.

A precondition of transformations is that inputs models conform to source metamodel. However, the transformation has to guarantee that the generated output models conform to target metamodel. As a consequence, the relationship between a transformation and its source metamodel is different from the relationship between the transformation and its target metamodel. We call those relationships *domain* and *codomain* respectively and define domain conformance in terms of them.

The domain of a transformation is the set of elements involved in the source patterns (also called left hand side parts) of the transformation. Similarly, the codomain of a transformation is the set of elements involved in the target patterns (also called right hand side parts) of the transformation. There is an additional restriction: models produced by the transformation should conforms target metamodel. A transformation $T$ and its input and output metamodels (resp. $MM_1$ and $MM_2$) respect the domain conformance relationship if the domain of $T$ is a submetamodel\(^1\) of $MM_1$, if the codomain of $T$ is a submetamodel of $MM_2$ and if all the well-formedness constraints defined in $MM_2$ concerning concepts present in both $MM_2$ and the codomain of $T$ also exist in the codomain of $T$.

## 4 Transformation Migration

Transformation migration should be performed by keeping in mind three phases: 1) impact detection, 2) impact analysis, and 3) transformation adaptation. During impact detection, the transformation inconsistencies caused by metamodel evolution are identified i.e. where transformation does not satisfy domain conformance. During impact analysis, the set of transformation updates needed to re-establish domain conformance is obtained possibly by using human assistance. Finally, during transformation adaptation, updates found in step two are applied. In this paper we present some ideas to perform the impact analysis phase. Those ideas are based in some preliminary experiments performed on some MDE-based frameworks that are quite sensible to transformation changes.

### 4.1 Impact Analysis

As we said above, impact analysis phase aims to identify the transformation updates needed to re-establish domain conformance. Our proposal relies on potential modifications. The current supported set is presented below. According to the kind of change applied on the source or target metamodel elements, suggestions are proposed. Note that some cases may need human intervention.

1. **Rename Class/Property**: All the occurrences of the renamed class/property should be updated in the PIT model by changing its name.

\(^1\) in the sense of typing defined by Steel *et al.* in [8]
2. **Move Property:** If this change occurs in the *source metamodel*, the path to access the property should be updated in all statements involving it. If the change occurs in the *target metamodel*, it is necessary to move the statement in the rule creating the new owner of the property and the path to access the other elements involved in the statement have to be updated.

If *property* is moved *between classes that are in the same transformation pattern*, then the path of the property should be updated.

3. **Modify Property:** No suggestions in this case because it is impossible *a priori* to ensure the typing of property. Furthermore, in case of multiplicity, the management of a unique element may become a collection. The impact should be resolved manually.

4. **Introduce Class:** Introducing a class in the *source metamodel* does not affect domain conformance but metamodel coverage of transformation. Hence, the propagation of this change depends of the purpose of the transformation and cannot be automatized.

Introducing a class in the *target metamodel* affects domain conformance only if this class is mandatory (*i.e.* referred with multiplicities “1”, “1..*”, “1..n”).

This change cannot be automatically propagated. Hence, the user should do it manually by choosing one of two options: 1) write a new transformation rule that creates elements conform to the new class based on concepts of source metamodel and modify an existing rule to call the former. 2) modify the target pattern of an existing transformation rule (mostly the one creating the owner of the added class) by including the new class on it.

5. **Introduce Property:** The same analysis than class addition can be done except that statement and not rule are added. This statement should be either in the class containing the property or in its subclasses. If the property is mandatory (*i.e.* multiplicity: "1", "1..*", or "1..n") this action has to be done.

6. **Replace Class:** All the occurrences of the replaced class should be changed by the new one. All the statements involving the properties of the replaced class should be fixed manually by finding its equivalence in the new one.

7. **Eliminate Class:** If the removed class belonged to the *target metamodel* and was the root of a pattern, the corresponding rule has to be removed. Otherwise, all the statements using the class or its properties should be removed. If after that elimination some patterns become empty, the corresponding transformation rule should be eliminated.

8. **Eliminate Property:** All the statements involving the removed property should be removed.

9. **Introduce Inheritance:** The same analysis than introduce property can be done except that the mandatory inherited properties have to be included to the transformation rules creating the subclasses if they not already exist in the rule creating the super class.

10. **Eliminate Inheritance:** All the statements that involves inherited properties by subclasses should be removed.
5 Example

We illustrate our approach using the well-known transformation example: from class to relational database model. The relational database model conforms to the RDBS metamodel simplification used in [1]. An original version of this transformation can be found in listing 1.1.

```java
1 mapping UML::Class::ClassToTable() : RDBS : Table {
2   name := self.name;
3   columns += self.features[UML::Attribute]->map DataTypeAttributeToColumn();
}|

6 mapping UML::DataTypeToType() : RDBS : Type {
7   name := self.name;
}|

10 mapping UML::Attribute::DataTypeAttributeToColumn() : RDBS : Column
11   when {
12     self.type.oclIsTypeOf(UML::DataType) and
13     self.multiplicity.range.upper -> first() = 1}{
14       name := self.name;
15       type:= self.type.map DataTypeToType();
}|

17 mapping UML::Attribute::MultiValuedDataTypeAttributeToTable() : RDBS : Table
18   when {
19     self.type.oclIsTypeOf(UML::DataType) and
20     self.multiplicity.range.upper -> first() <> 1}{
21       name := self.name;
22   }|
```

Listing 1.1. Transformation: From Class to Relational

The managed changes refer to the evolution of UML from version 1.5 to version 2.0. An overview of this evolution can be found in [6]. We nevertheless slightly modify it in order to manage multiplicity. The following list gathers some of the changes.

- **Eliminate Property:** Property "multiplicity" is **eliminated** from "StructuralFeature".
- **Rename Class/Property:** In "Class", the property "features" is renamed to "attributes".
- **Modify Property:** In "Class", Property "attributes" is modified by changing its type from "StructuralFeature" to "Attribute".
- **Replace Class:** Class "Attribute" is **replaced** by class "Property".

**Impact Analysis:** According to the applied changes and the implied elements, the migrations to performed can be deduced from the suggestion list, we provide in section 4.1:

- All the statements where the `multiplicity` property appears are removed: **line 12:** self.multiplicity.range.upper -> fist() = 1 in `DataTypeAttributeToColumn` rule and **line 19:** self.multiplicity.range.upper ->fist ()<>1 in `MultiValuedDataTypeAttributeToTable`.
- In all the statements, the `features` property is renamed into `attributes` e.g. in **line 3:** columns+=self.features [UML::Attribute]->map `DataTypeAttributeToColumn()`
The typing issue relative to the attributes property has to be solved by hand such as in line 3: columns=+self.attributes [UML::Attribute]->map Data-TypeAttributeToColumn() in the ClassToTable rule.

6 Conclusions and Future Work

In this paper, we have introduced the notion of domain conformance, the relationship between a transformation and its metamodels. We have also studied how a metamodel evolution may affect that relationship and proposed a transformation migration process based on a first set of actions to re-establish domain conformance. Our future works are twofold. The presented set of actions is the basis. We need to refine it by further scrutinizing metamodel evolutions (e.g. relative to the inheritance issue) and possibly add new transformation migrations suggestions. Secondly, we want to study the way to apply these actions, for example using a high order transformation, operators, or manually, since user’s intervention is required.

7 Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the ‘Contrat de Projets Etat Region (CPER) 2007-2013’

References

Towards Metamodel Evolution of EMF Models with Henshin

Stefan Jurack\textsuperscript{1}, Florian Mantz\textsuperscript{2}

\textsuperscript{1} Philipps-University of Marburg, Germany, sjurack@mathematik.uni-marburg.de
\textsuperscript{2} Bergen University College, Norway, fma@hib.no

Abstract. The Eclipse Modeling Framework (EMF) has become a widely used technology to model domain-specific languages by means of metamodels. The evolution of such metamodels requires the co-adaption of its instances and possibly other related artifacts which is a considerable challenge. In particular, if model migration is performed in-place, i.e. by changing models itself, conformance between metamodel and its instances need to be ensured permanently throughout the migration process. In this paper, we present an in-place transformation approach towards metamodel evolution based on the EMF model transformation tool Henshin. The transformation approach is presented by means of an example: The evolution of a Petri net metamodel, its instances and related transformations.

1 Introduction

Model evolution has become an interesting research field since models play a major role in software engineering. In particular, the evolution of metamodels is a considerable challenge of modern software development as changes may require the migration of its instances. Work in this direction exists already, e.g. in \cite{1} Cicchetti et al. categorize different model changes according to their impact on instances and their resolvability, while the authors in \cite{2} analyze a number of semi-/automatic approaches towards model and meta-model co-evolution.

In terms of model-based and model-driven development the Eclipse Modeling Framework \cite{3} is a well-known and widely used technology to define structured data models, i.e. in particular domain-specific (modeling) languages, providing code generation facilities as well. EMF’s core language, Ecore, complies with Essential MOF (EMOF) as part of OMG’s Meta Object Facility (MOF) 2.0 specification \cite{4}. The question arises, how EMF metamodels and their instances can be co-/evolved. A number of approaches, e.g. Epsilon Flock \cite{5}, ATL \cite{5} use an out-place transformation approach, i.e. separate models are created during transformation.

In this paper, we present an in-place metamodel evolution approach based on the EMF model transformation tool Henshin \cite{6}. Its transformation language is based on graph transformation concepts and contains a control structure concept called transformation units. Furthermore, a Java(Script)-based computation of attributes is available.

The paper is structured as follows: An example metamodel evolution is shown in Sec. 2 based on simple Petri nets. Section 3 presents a first approach to adapt transformations related to an evolving metamodel. A discussion of related work including a comparison to other approaches and concluding remarks can be found in Sec. 4 and 5.
2 Evolution and co-evolution with Henshin

This section presents our metamodel evolution approach with Henshin illustrated by means of a Petri net example. At first, we describe the evolution scenario and show the metamodel to be evolved at its states before and after evolution. Afterwards, corresponding Henshin rules and concepts are explained.

In order to support the reader, Fig. 1 shows a brief overview of the scenarios presented in this section and Sec. 3, with involved models and interrelations. Figure identifiers next to some model nodes relate to figures in this paper and shall help to keep track of referred models.

Consider an EMF Petri net metamodel as shown in Fig. 2, consisting of a Place connected by Transitions and a root node Net. Complying Petri net models can be deduced easily and are not shown due to space constraints. In our scenario we want to replace the direct edges src and dst between Transition and Place by concrete nodes representing this interrelation, e.g. allow for weighted edges in a next step. An accordingly evolved metamodel with appropriately deduced node and edge names is shown in Fig. 3. Please note that Fig. 2 and Fig. 3 show the concrete syntax of EMF models at which each node, attribute and reference represents an instance of Ecore type EClass, EAttribute and EReference, respectively. Since EMF models have to comply with its metamodel permanently, an evolution using in-place transformations has to be basically performed in three steps: First of all, new required elements have to be introduced into the metamodel. In the second step related changes at model level have to be performed, e.g. delete/change old structures and create new ones. Finally, outdated and at this point unused metamodel elements are deleted.
The following Henshin rules realize our scenario generically, i.e. each rule may be applied to any EMF models for the use case “replace edge by node” (edge2node). These rules are reusable and can be considered as metamodel refactorings. In contrast to conventional illustrations, rules are shown in a space-saving integrated manner, i.e. elements in the LHS and RHS being mapped appear as preserve, while element occurring either in a RHS or LHS appear as create or delete, respectively. In addition, rules may be equipped with parameters, used as object identifier or attribute value. If passed to a rule before its application, they confine possible matches. Other parameters are set automatically during matching and may provide matched objects and values afterwards.

Evolution rule AddMMRefClass in Fig. 4 represents the first step of our evolution and creates the metamodel elements which shall replace a given edge. Only parameters nameSrcNode and nameTrgNode were initially set to “Place” and “Transition”, while the edge (being finally replaced) is matched non-deterministically. Consequently, to evolve src and dst two evolutions have to be performed. Please note the parameters in the create-part, setting attribute values dependent on the match of preserved elements.

Step (2) in our evolution scenario is trickier, since models and metamodels are located on different layers. To overcome this gap we exploit Henshin’s capability to transform/create any EMF model, i.e. Henshin models as well. In fact, migration rule MigrateModels shown in Fig. 5 creates another Henshin transformation system mainly consisting of a Rule with LHS (upper area), RHS (lower area), Mappings to specify preserved elements, etc., in order to replace an edge with a node and two edges at model level. The elements in the center are pre-defined by the matching of the previous rule and used as typing for Nodes and Edges of the generated co-evolution rule. Equal names mean equally matched objects. IndependentUnit and CountedUnit care for an application as often as possible to cover all occurrences. Note that the migration rule is typed over Henshin’s transformation system metamodel [6,7] and Ecore, while the resulting co-evolution rule is typed over our Petri net metamodel.

The evolution rule performing the third step, that is deleting the substituted edge in the metamodel, is simple and left out here. It requires one parameter only: the edge to be deleted, objOldRef, which is available by the matching of the first rule.

Fig. 4. Evolution Rule AddMMRefClass: Creates an edge node in the metamodel.
Fig. 5. Migration Rule MigrateModels: Creates the co-evolution rule.

3 Evolution of Henshin transformations

Besides models, metamodels are often referred to by other artifacts such as transformation rules, e.g. specifying model changes or implementing checks over models. If a metamodel evolves, related artifacts need to be adapted appropriately. Theoretical foundations of applying rules to rules have already been explored to some extend by Parisi-Presicce in [8] and Ehrig et al. in [9].

Consider for example a Henshin rule set for detecting cycles of arbitrary path length in a Petri net by constructing the transitive closure of places connected by transitions and evaluating this afterwards. Henshin rule TransitiveClosure\(^3\) in Fig. 6 performs the former. The co-evolution of this rule according to the scenario in the previous section takes place in step (2), analog to the co-evolution of models presented above. Again, Henshin itself is utilized to transform its rules and the rule co-evolution rules are also specified in a generic manner. This means they can be used to co-evolve every Henshin rule in terms of our metamodel evolution edge2node.

Fig. 6. Henshin Rule TransitiveClosure: Creates additional transitions in a Petri net.

\(^3\) This rule contains a further stereotype, forbid, which specifies parts of a negative application conditions (NAC), i.e. what is not allowed to occur in the context of a match.
Figure 5 in the previous section reveals essential parts of the abstract syntax of Henshin transformation systems and its rules. Please note in particular that in rules nodes may be mapped or not. Note furthermore that edges are not mapped at all, as EMF supports at most one edge per type between source and target node, i.e. the edge mapping is determined by the mapping of its source and target nodes. Since we replace edges with a node (and two edges), we may have to introduce a mapping for the newly created node as well. Accordingly, the migration is performed in two phases: In the first phase related edges in all graphs, e.g. LHS and RHS, are replaced while, if required, the mapping of the created nodes is added in the second phase. Figure 7 shows the rule co-evolution rule realizing the first phase. The upper area shows the deletion of the old edge while the lower area shows the creation of the intermediate node and its edges to source and target. Equal names mean equally matched objects, again, according to the application of rule AddMMRefClass shown in Fig. 4. The second phase is omitted here due to space limitations. Its rule co-evolution rule creates the mappings between newly created intermediate nodes in related graphs if source and target nodes are also mapped.

Both rules are applied in a sequence each as often as possible by Henshin’s control concepts.

![Rule co-evolution rule: Replace an edge by a node in a Henshin rule.](image)

### 4 Related work

AGG [10] bases on graph transformation similar to Henshin, but uses a concept of fixed type graphs which the authors consider not suitable for metamodel evolution.

In [11], Sprinkle introduces a visual graph transformation based language for metamodel evolution that allows to specify the differences between two metamodels rather their similarities. However, this language does not provide any support for reuse and also performs out-place transformations since XSLT is used under the hood. Henshin provides a visual syntax, too, which is however not dedicated to metamodel evolution.

In [12], the authors present an metamodel evolution approach based on COPE. As Henshin, it uses in-place transformations. A general difficulty of in-place transformations is that models and metamodels need to be changed in a coordinated manner as
shown above, to guarantee a permanent compliance with their metamodel. However, COPE decouples models and metamodels during transformation and therefore cannot guarantee a successful model migration. Herrmannsdörfer et al. state that reusability and expressiveness are important features for a metamodel evolution approach. Transformation rules in our solution generically defined, thus highly reusable. Due to the rich set of control structures Henshin can be considered expressive as well.

In [5], Rose et al. compare their model migration language Epsilon Flock with other approaches, namely Ecore2Ecore, ATL and COPE and check for certain features as “Automatic copy”, “Automatic unset” and supported “Modeling technologies”. Except COPE, all approaches in [5] use out-place transformations. A detailed comparison between in-place approaches in particular, including Henshin, would be interesting.

5 Conclusion

Henshin has been applied successfully to metamodel evolution by a first example. Moreover, related rules has been transformed accordingly as well. This shows, that Henshin provides a feasible approach towards metamodel evolution while, however, additional tool support would be reasonable in this domain.

References

RCVDiff - a stand-alone tool for representation, calculation and visualization of model differences

Mark van den Brand  
Eindhoven University of Technology  
Den Dolech 2  
5612 AZ Eindhoven  
m.g.j.v.d.brand@tue.nl

Zvezdan Protić  
Eindhoven University of Technology  
Den Dolech 2  
5612 AZ Eindhoven  
z.protic@tue.nl

Tom Verhoeff  
Eindhoven University of Technology  
Den Dolech 2  
5612 AZ Eindhoven  
t.verhoeff@tue.nl

ABSTRACT

In model comparison one can distinguish three major sub-problems: representation, calculation, and visualization of model differences. Existing stand-alone tools for managing model differences usually focus on one or sometimes two aspects of model comparison. In this paper we discuss our stand-alone tool called RCVDiff, that incorporates all three aspects of model comparison. Furthermore, we comment on some issues in the process of model comparison that are not easily perceived if the three aspects of model comparison are considered independently.

Keywords

Metamodeling, Model comparison, Model differences

1. INTRODUCTION

Model Driven Software Engineering (MDSE) is a field of Software Engineering which focuses on models as main design artifacts, and uses model transformations as means of relating models. Consequently, mature model configuration management systems are required to manage the complexity of modeled systems in MDSE environments. One of the major functions of model configuration management systems is model comparison. Model comparison (model differencing) is a complex process which consists of three concerns: representation, calculation, and processing of differences [9]. The rationale behind this separation of concerns is that usually it is not only required to calculate differences, but it is required to store, process, and visualize them in the context of a model configuration management system.

Usually, research in the area of model comparison is focused on only one aspect of model comparison. For example, model differences representation mechanisms are discussed in [4, 10, 7]. Model differences calculation is discussed in [6, 8, 5]. Model differences visualization is discussed in [16]. Consequently, tools that are developed to validate the research of a certain aspect, focus only on that specific aspect. This inevitably creates the problem of integrating different tools into one system for model comparison. This problem becomes even worse because these tools may use different metametamodels to describe their metamodels, models and model differences. Thus, (error-prone) transformations must be defined to transform metamodels, models and difference models used in one tool to their counterparts used in other tools.

In this paper we discuss the details of our tool that combines all three aspects of model comparison [3]. Our tool is based on our research on model differences representation and calculation [14], and model differences visualization [15]. Furthermore, this tool is stand-alone and can be easily adapted to be used as a part of a custom configuration management system or a larger CASE tool.

The theoretical basis of our tool is discussed in Section 2. Thereafter, in Section 3, a description of a static and a dynamic architecture of our tool is given. Next, in Section 4, a use-case scenario is presented, which describes the usage of our tool. Finally, in Section 5, we conclude the paper.

2. PRELIMINARIES

In this section we first describe our approach to the representation of model differences. In our approach, the differences are represented by differences models which conform to a differences metamodel, similarly to approaches like EMFCompare [2] or the one presented by Cicchetti et. al. [4]. Next, we briefly describe our approach to the calculation of model differences, that produces differences models that conform to the specified differences metamodel. The details of both approaches can be found in [14]. Finally, we describe our approach to visualization of model differences. The details of this approach can be found in [15].

2.1 Representation of model differences

Our approach for the representation of model differences allows those differences to be seamlessly used in modeling environments. Thus, the differences between two models are represented by a difference model which conforms to a differences metamodel. The differences metamodel is based
on the metamodel depicted in Figure 1.

This metamodel describes both metamodels and models. Metamodels are obtained by instantiating the Metamodel element, and models are obtained by instantiating the Model element. This metamodel can be considered as a domain specific metamodel which is geared towards representation of model differences, and not towards general modeling (like MOF or Ecore). Thus, this metamodel is comparable to the cores of MOF and Ecore.

The differences metamodel is an extension of the introduced metamodel and is depicted in Figure 2.

The differences models are instances of the Differences-Model element. The main building blocks of the differences models are instances of ChangedElement, DeletedElement, and AddedElement. Assuming that the differences model represents the differences between models $A$ and $B$, then the instances of the AddedElement are elements that are in model $B$ and not in model $A$, the instances of the DeletedElement are elements that are in model $A$ but not in model $B$, and the instances of the ChangedElement are elements that represent the same entity in both models but are not structurally identical.

### 2.2 Calculation of model differences

Traditional approaches for the calculation of model differences are based on tree-matching algorithms. These algorithms match the nodes of two trees that represent two models being compared, and based on this matching the differences are calculated. Several types of matching are recognized: static-identity, signature-based, similarity based or language-specific [9].

Our algorithm for calculating differences is also based on tree-matching algorithms. Unlike traditional approaches, that support only one type of matching, our algorithm is defined in such a way to support all four types of matching. In order to allow such a highly configurable calculation process, we extend our metamodel with additional elements. The extended metamodel is interpreted as a calculation metamodel and is depicted in Figure 3.

Calculation models are used by our algorithm and they have two important features. The first feature is represented by instances of the CalculationConfiguration element. This feature of the calculation model opens the possibility of specifying the metamodel-specific configurations that are used to influence the calculation process of models related to the specific metamodel. Thus, for all models that conform to a specific metamodel, only one calculation configuration needs to be set. The second feature is represented by instances of the ComparisonMElement. These instances are used instead of model elements to represent nodes in model trees.

An instance of a ComparisonMElement is generated for each model element in a preprocessing step, with the help of the metamodel-specific configuration.

#### 2.2.1 Model comparison algorithm

The input for our comparison algorithm are two models $A$ and $B$ and the metamodel-specific configuration. Our comparison algorithm consists of three steps. In the first step, the trees representing models are traversed bottom-up, and the similarities between objects in models $A$ and $B$ are calculated. We say that two objects are similar if they can be considered as the same entity. We define similarities by using a similarity function which returns true if two objects are similar, and false otherwise.

In the second step, based on the similarities found, a matching of objects is calculated. The matching is performed by traversing the tree top-down. At the first level, based on the similarities found, some objects may be matched. For all matched objects at the first level, the matching process continues recursively until the bottom of the tree is reached or there are no sub-objects that can be matched.

In the last step, the calculation of differences is done based on the matchings found.

### 2.3 Visualization of model differences

In our approach to visualization of model differences, we adopt the idea of information visualization proposed by Shneiderman [13]: Overview first, zoom and filter, then details-on-demand. The reason for adopting this idea is based on the fact that model differences are “information content” that need to be visualized. Thus, it is required to have overview capabilities, such that the global meaning of differences can be comprehended. Next, it should be possible to zoom in and filter differences, such that the users of configuration management systems can syntactically and semantically associate the differences to the parts of the models that those differences are related to. Finally, the selected differences should be rendered by using a sufficient level of detail to help the users understand them better.
Traditional (e.g. text-based, tree-based or diagrammatic) approaches to visualization of model differences, if considered in separation, do not fully support the idea of information visualization [15]. Thus, in order to fully support this idea, we combine two visualization approaches.

The first approach we use are polymetric views, first described in [11]. The polymetric view is a lightweight graphical component for visualizing a set of related entities. This is accomplished by defining metrics that can be applied to the set of entities that is to be visualized, and by specifying a view. The view is specified by relating defined metrics to the graphical attributes of shapes that will represent entities. Thus, by calculating the metrics on elements of the set, and by inferring the values of graphical attributes of the shapes that will represent entities, the entities can be visualized on a graphical canvas. In the context of models, the metrics are based on metamodel elements. Consequently, they can be calculated for model elements conforming to those metamodel elements. This allows a visualization of model elements by using the calculated metrics.

The second approach we use is a framework for visualization of metamodel based languages (MMVisualizer in further text). MMVisualizer uses a declarative approach for the visualization of models. In order to visualize a model, the user needs to specify a set of rules. Each rule maps one metamodel element (of the metamodel that the model conforms to), to a graphical shape. Based on a type of the used rule, a predefined shape (e.g. rectangle, oval, line,...) is used to visualize model elements conforming to the mapped metamodel element. Although the rules are designed such that predefined positioning information can be used for actual positioning of model elements, this is not required per-se, but the layout of a model is calculated by using the dot [1] framework.

In order to use the two specified approaches the old model and the differences model are combined in one unified model.

In the unified model the model differences are related to, and thus in a sense annotate, model elements. Thus, the metrics can be calculated on the unified model by also considering the model differences. Furthermore, by examining the unified model, model elements can be colored appropriately in MMVisualizer (deleted elements to the old model are colored red, added elements to the new model are colored green, and changed elements are colored blue).

3. TOOL ARCHITECTURE

This section gives an overview of the architecture of the RCVDiff tool. The tool has been implemented in the Java programming language. A logical view of the tool architecture is provided by the package dependencies depicted in Figure 4. The package hierarchy contains three high-

![Figure 2: Differences metamodel](image)

![Figure 4: RCVDiff package hierarchy](image)
V contains tool components related to the visualization of model differences.

The high-level data flow diagram of the RCVDiff tool is depicted in Figure 5. The tool has two external "access points". One access point is a differences calculator. The differences calculator receives an old and a new (evolved) model, their common metamodel, and a configuration file, and produces a differences model. The resulting differences model can be processed by other tools, or can be used as input to a differences visualizer, which is the second access point of the tool. The differences visualizer must also receive an old model, a metamodel, a configuration file for polymetric views and a configuration file for MMVisualizer.

4. TOOL USE CASE

This section provides a small use-case in which two designers work on the same model consecutively, and wish to compare their consecutive versions of that model. Assume that the first designer has created a state machine model $A$ depicted in Figure 6, and that the second designer has changed that model to a model $A'$ depicted in Figure 7.

![Figure 5: High-level data flow diagram of the RCVDiff tool](image)

![Figure 6: Example old model](image)

The metamodel of models $A$ and $A'$ is the same and is depicted in Figure 8. Both models $A$ and $A'$, as well as their metamodel $M_A$, conform to the metamodel depicted in Figure 1.

Notice that both models, as well as their metamodel, are represented in the form of a tree in order to reflect the fact that, at this point, the visualization aspects have not yet been defined. Also, notice that we use $LID$ attributes of metamodel and model elements to denote locally unique identifiers of those elements. These identifiers can be supplied by the tool, or can be automatically generated (we have devised a procedure for the automatic generation of locally unique identifiers that assigns an identical identifier to the same metamodel or model element each time it is invoked on the same metamodel or model [12]). Furthermore, the attributes and references of metamodel elements are assigned (attribute or reference) identifiers as well. The existence of these identifiers is crucial in approaches to model versioning which rely on references as a way of relating elements of differences models to elements of models. Thus, the locally unique identifiers of model elements are used as references of these elements in the differences model. Moreover, model elements reference their conforming metamodel elements and thus metamodel elements must also be equipped with identifiers. Furthermore, identifiers of metamodel elements, attributes and references are also used in the calculation and visualization configuration files.

Next, assume that the first designer would like to inspect the changes to his model in the new model. In order to do that, he must compare the old and the new model, and visualize the obtained model differences.

Our difference calculation tool initially uses a predefined metamodel-independent calculation configuration, however this configuration can (and should) be changed by a domain expert to obtain more accurate results. Each configuration is specific for a metamodel, and thus can be used in comparison of all pairs of models conforming to that specific metamodel. An example of the configuration, specific for the example metamodel $M_A$, is given in Listing 1:

```xml
<CalculationConfiguration CID="0" metamodelid="SMMM">
  <ComparisonMMElement MMElementid="0" name="State" referencedID="1" RID="1" type="string"/>
  <ComparisonMMElement MMElementid="1" name="Transition" referencedID="2" RID="2" type="string"/>
  <ComparisonMMAttribute AID="0.1" name="name" referencedID="1" referencesThreshold="0.5" referencesThreshold="0.5" subobjectsThreshold="0.5" subobjectsThreshold="0.5" />
  <ComparisonMMAttribute AID="1.1" name="name" referencedID="1" referencesThreshold="0.5" referencesThreshold="0.5" subobjectsThreshold="0.5" subobjectsThreshold="0.5" />
</CalculationConfiguration>
```
The calculation configuration of Listing 1 contains three instances of a ComparisonMMElement since there are three instances of an MMElement in the metamodel. The instance of a ComparisonMMElement with MMElementid attribute having the value 0 is used to guide the comparison algorithm when comparing the instances of the metamodel element with LID 0 (i.e. state machines). Since state machines have one attribute (name, having an AID 0.1), the ComparisonMMElement used in comparing state machines contains one instance of the ComparisonMMAttribute, related to that attribute. This attribute is set to be the key, thus two state machines that have identical names are considered the same.

The instance of a ComparisonMMElement with MMElementid attribute having the value 1 is used to guide the comparison algorithm when comparing the instances of the metamodel element with LID 1 (i.e. states). Since states, like state machines, have one attribute, the ComparisonMMElement used in comparing states contains one instance of the ComparisonMMAttribute. This ComparisonMMAttribute uses a comparison function named compareTwoStrings_Identical, which considers two state names to be equal if they are identical. Any other user-defined function could be used instead of the used function, the only requirement is that the comparison function must take two strings as arguments, and must return a value between 0 and 1. Then, a configured threshold is used to determine if the two compared attributes are equal.

The instance of a ComparisonMMElement with MMElementid attribute having the value 2 is used to guide the comparison algorithm when comparing the instances of the metamodel element with LID 2 (i.e. transitions). Since transitions, like state machines and states, have one attribute, the ComparisonMMElement used in comparing transitions contains one instance of the ComparisonMMAttribute. This ComparisonMMAttribute also uses a comparison function named compareTwoStrings_Identical, which considers two transition names to be equal if they are identical.

Next, in order to visualize the differences in a metamodel-specific way, a mapping between the metamodel model and the dot shapes needs to be defined. A mapping consists of a set of rules. Each rule is used to map one metamodel element to one dot shape. An example mapping is given in Listing 2:

Listing 2: Mapping for visualization of model differences by using MMVisualizer

```xml
<ComparisonMMElement MMElementid="1" attrsimexpression="" externalsimfunction="" referencesThreshold="0.5f" subobjectsThreshold="0.5f">
<ComparisonMMAttribute attributeID="1.1" key="false" used="true" group="0">
  functionName="compareTwoStrings_Identical"
  threshold="0.5" weight="1"/>
</ComparisonMMElement>

<ComparisonMMElement MMElementid="2" attrsimexpression="" externalsimfunction="" referencesThreshold="0.5f" subobjectsThreshold="0.5f">
</ComparisonMMElement>
</CalculationConfiguration>
```

Listing 3: Custom view

A custom view in Listing 3 specifies that all the model elements should be represented as a tree (LAYOUT: Tree). The length of an element is set to the number of attributes of the element (LENGTH: AttributesNumber). The width of an element is set to the number of references of the element (WIDTH: ReferencesNumber). The color of an element is set to red hue based on the number of changes to the element (COLOR: ChangesNumber). The outline color is set based on the custom metric named MyCustomMetric, and the elements are sorted by their color (OUTLINE: MyCustomMetric, SORT: Color).

We will now assume that the differences between models $A$ and $A'$ have been calculated in the calculation part of the tool by using the example calculation configuration presented in this section. Then, the initial result of the visualization part of the tool, using the calculated differences model, and the example configuration files for MMVisualizer and polymetric views, is depicted in Figure 9. The initial result contains the differences represented by using a GLOBALTREEVIEW view. Next, if a user clicks the most red element (representing the state machine), the MMVisualizer is activated and produces the result depicted in Figure 10.
5. CONCLUSION

In this paper we described our tool that handles representation, calculation and visualization of model differences. This tool provides a stand-alone Java-based framework for dealing with model differences. This kind of tool is needed because existing tools either deal with only one aspect of model differences (and thus it is hard to combine them), or they are too tightly integrated into a larger CASE tool in order to be easily used in a model configuration management system. We designed our tool to be generic in the following ways: First, it allows the users to use all graph-based metamodels and conforming models. Next, it allows for a highly customizable model comparison process. Furthermore, the results of a model comparison are completely metamodel-independent model differences, represented as models conforming to a differences metamodel, which can be processed further. Finally, our tool allows for a user-guided visualization of model differences. This is achieved by incorporating polymetric views in a framework for visualization of metamodel-based languages.

6. REFERENCES

The Case for Batch Merge of Models – Issues and Challenges

Lars Bendix¹, Maximilian Koege⁵, Antonio Martini³

¹ Department of Computer Science, Lund University, S-221 00 Lund, Sweden
² Institut für Informatik, Technische Universität München, D-85748 Garching, Germany
³ Dipartimento di Informatica, Università di Parma, I-43100 Parma, Italy

bendix@cs.lth.se, koegel@in.tum.de, anto@ideazan.it

Abstract. Modelling is a fairly mature technology that is already being adopted by industry. Unfortunately the tools for supporting Model-Driven Engineering are not at the same level of maturity. In particular merge tools that are an important help for collaboration lack a lot to be desired in providing the support that is needed on industrial projects. Current model merge tools and research approaches focus on interactive merging where the developer resolves possible merge conflicts one by one during the merge process. However, developers are used to merge tools from the textual domain that work in a batch mode – the merge tool is provided with three files as input and produces a file with the proposed merge result – including possible unresolved merge conflicts. The developer can then pick up this result and resolve possible conflicts at a time and in an order that pleases him. In this paper, we motivate why research should start to focus also on batch merge for models and we describe how it could work. Furthermore, we sketch the research agenda that is needed to address the issues and challenges in realizing batch merge for models.

Keywords: Model merge, interactive vs. batch mode, conflict representation, conflict management, issues and challenges, research agenda.

1 Introduction

In a software development project, models are important artefacts and they have become more and more important in recent years, especially with the adoption of Model-Driven Engineering. Working with and evolving models in an industrial context is a highly collaborative activity where multiple developers participate. Therefore it is important to be able to manage changes to the models and to manage conflicting changes carried out by different developers working in parallel. The activity of integrating such changes into a consolidated model is called merging and involves resolving conflicting changes to build a new resulting model that reflects the intentions of all involved developers as far as possible.

The road that model merging has taken so far differs from the way in which merge works in a traditional textual context. The goal of model merge is to produce the “perfect merge result” where all conflicts are resolved. When the merge tool encounters a
conflict that cannot be resolved automatically, it asks the developer to resolve the conflict – usually constraining the choice to one of the two alternatives. The advantage is that new work can continue immediately after the merge; the drawback is that we may force the developer to make a lot of decisions up-front. Traditional textual merge, on the other hand, has no illusions of perfection. They happily ignore syntax and semantics of what is being merged and whenever a conflict is encountered it is just marked in the text. The advantage is that a merge result is produced very quickly without any user intervention; the drawback is that the developer should put the result right (or at least check if it is right) before continuing his work.

We have experienced a great desire from industry to be able to work with model merge in the same flexible way as when working with merge for traditional programming. In the following, we look at the further motivation for batch merge of models, then we analyse what issues and challenges need to be addressed. Finally, we briefly discuss some existing research and draw our conclusions.

2 Motivation

In practical software development it often happens that work is carried out in parallel for many different reasons. Traditionally parallel work is handled by selecting a suitable branching strategy [1] that will support the specific type of parallel development that is carried out. However, whenever we talk about branching out to several lines of development, we also create the “double maintenance problem” [2] which means that changes to one branch should also – sooner or later – be done to the other branch(es) to keep them synchronized. Integrating changes between branches is done using a merge tool and is a very frequent activity.

From a previous study of model merge in industry [4], it turned out that, besides quality issues with the results produced by the merge tools, a major complaint was that the developers found interactive merge an awkward and disruptive way of working. The fact that there might be a long session of resolving conflicts before the merge would be done meant that there was a perceived high mental cost involved in starting a merge, thus discouraging people from doing it as frequent as they might have wanted. Sometimes the right choice was not one of the alternatives, but a mixture, which was not possible. There would also be occasions where they would regret a previous choice later on in the process, and in certain situations they would need to consult another person – that was not there – to resolve the conflict.

This leads us to propose a “new” merge strategy for model merge – batch merge – where the total cost of the merge process can be split up into two “stages” (merge production and conflict resolution) that can be carried out independently and when it is convenient for the developer. Even in the case where conflict resolution is initiated immediately after the merge production, batch merge has advantages. Interactive merge forces developers into resolving conflicts in a particular order – that of the tool. Batch merge would allow for “sequence independent conflict resolution”. Batch merge is not “new” as it is the way that textual merge is done and we would like to see something similar for model merge.
We will now express these ideas in a number of more detailed scenarios. These scenarios came up during interviews and discussions with developers and from drawing on our own experience:

*Produce a merge* – the two alternatives and the common ancestor of the system (often consisting of several files) are fed into the merge production tool. It produces a system where the conflicts are marked and the conflicting alternatives are represented in the model.

*Discuss conflicts in asynchronous collaboration* – you need to discuss a conflict and collaborate with someone else to figure out how to resolve it. You can either postpone the resolution until that person is present or you can send the merge result to the person and ask him to resolve the conflict.

*Resolve conflict* – working from a merge result, it is possible to allow the developer to choose the order in which conflicts are resolved. Furthermore, it is possible to take a more global view of the conflicts and discover connected conflicts that have to be resolved as one logical entity.

*Virtual merge* – shows you what happens if you merge, but does not do the actual merge. You only want to have an overview of where conflicts and changes are (like you can do in the Ragnarok system [5]) to be aware of what is happening and do not want to have to interactively “resolve conflicts” first.

### 3 Issues and challenges

During our analysis of issues and challenges in providing batch model merge we identified many different topics. For presentation reasons, we have grouped them into two categories – merge production and conflict management – and for lack of space, we state only the most important ones (and leave out those that do not fit into these two categories).

#### 3.1 Merge production

For the production of a merge result there are some things like model matching and conflict detection that are common for both interactive merge and batch merge. However, in batch merge production there is no conflict resolution integrated. The merge production is fully automated and there is no possibility to ask for developer intervention. The consequence of this is that there are two major differences between batch and interactive merge: the produced output might not be a valid model and that we will have to mark and represent conflicts in some way. Besides that there are a number of minor particularities.

*Conflict mark-up* – in the case where there have been concurrent changes to the same Unit of Comparison (UC) we have a conflict. There may be cases in which a careful analysis will be able to produce a correct result, but in general there is no way to guarantee a correct result. Interactive merge relies on “human intelligence” from the developer intervention to produce a correct result. In batch merge we will have to give up and signal a conflict for the UC in question. This conflict will have to be
Alternative representation – when a conflict is detected it means that the same UC has been changed in two different ways. Besides signalling the conflict in the result we also need to incorporate information about what the two alternatives are. This will help the developer to resolve the conflict later on with the same information available as for interactive merge.

Violation handling – for textual merge the question of when there is a conflict is simple: when there are concurrent changes to the same UC. Since textual merge ignores the possible syntax and semantics of what is merged, it also ignores potential problems with the syntax and semantics of the produced result. This differs completely from the approach in interactive model merge where there are a number of constraints on what is a legal output. This means that interactive model merge talks about hard and soft conflicts – hard conflicts being cases where the result breaks the meta-model constraints; soft conflicts being cases where the result breaks other validation constraints, but still conforms with the meta-model. A merge result with soft conflicts will load in the model editor, whereas a result containing hard conflicts will not. Therefore interactive model merge goes to great length to discover and avoid hard conflicts. For batch merge it is doubtful whether this should be considered part of merge result production since it requires more analysis and work than just “simple” conflict detection at the level of the same UC. Furthermore, batch merge already has to handle merge results that might not conform to the meta-model, so this might be considered just one more thing that should be taken into consideration when designing the meta-model for the merge result.

Change mark-up – when a UC is changed by only one of the parties, it should be straightforward to include the changed UC in the merge result. However, as it was pointed out above, that might lead to cases where the merge result violates some constraint imposed either by the meta-model or in other ways. Such violations will have to be discovered and sorted out in the subsequent conflict management phase. Furthermore, information about other changes may be useful when the developer has to resolve real merge conflicts. For these reasons we suggest to consider the mark-up also of “simple” changes to avoid loss of information from the merge production phase to the conflict management phase. This information could be obtained from a diff functionality, but it seems better to just put it in the result right away.

Merge result format – interactive model merge can count on developer intervention for sorting out conflicts and can thus produce a merge result that conforms to the meta-model and is ready to use. For batch merge we do not have that possibility. We will have unresolved conflicts that need to be marked up in the result. We will have to represent the alternatives in the case of a conflict. If we chose to ignore validation constraints in the merge production phase, we will have to deal with merge results that might not conform to the meta-model for what we are trying to merge. Finally, we would like to mark up all changes (also those that do not create conflicts) to avoid loss of information. This means that we have to come up with a format or model merge meta-model that is sufficiently robust and complete to be able to represent all these possibilities for the merge result. Then the subsequent conflict management phase can be used to gradually bring the merge result in a state that it conforms to the original meta-model – and possible additional validation constraints. How such a
model merge meta-model should look like and what are the requirements (in particular for the violation handling aspects) is a widely open question.

Merge input format – since the merge production phase creates a result that may not respect the meta-model, we have to consider what should happen if a developer uses the result as the input to a new merge without – or before – bringing it in conformance with the meta-model. The ideal solution would be a merge production that is able to handle not just input that conforms to the meta-model, but also inputs that break the meta-model but still conform to the model merge meta-model. The advantages of such a solution are clear, however, the difficulties in implementing such a merge are far from clear.

3.2 Conflict management

The result that is created by batch merge is not guaranteed to be correct or work – rather it is almost guaranteed to be inconsistent in one or more ways with respect to the developer’s intentions and expectations. In the second phase of the merge process – conflict management – the developer can work on the produced merge result to bring it in a state that is satisfactory for him and reflects his intentions. He needs to inspect and change the merge result, to resolve conflicts and other problems with the help of functionality that can analyse the merge result, and to continuously validate the result to discover remaining problems.

Visualization and editing – we will need new tools to visualize and edit the merge result produced by batch merge since it will probably not conform to the meta-model. However, when a standard model merge meta-model is established, it should be no problem to build such tools. Current model editors ignore visualization aspects of model merge – probably because there are no such aspects to visualize when interactive merge is used. However, as it was discussed above for virtual merge, visualization can be of great help for creating awareness about how the system changes.

Model validation – since the merge result does not conform to the meta-model, it is very important to be able to validate the resulting model at any given time. The validation process should be simple and quick, and the validation tool should be able to work with the model merge meta-model and point out where and what does not conform to the original meta-model.

Connected conflicts discovery – some conflicts (and/or changes) may be connected/related. The prime example is a refactoring. In batch merge all changes and conflicts are directly represented in the result and an intelligent tool can analyse the result to find connected/related conflicts/changes so they can be presented – and resolved – together. A special case of this is for operation-based merge, where conflicts have options, options can be conflicting with the options of other conflicts or they can require a certain option of another conflict.

Rich alternative proposal – since we do not have to respect the original meta-model – only the model merge meta-model – we can propose more options for conflicts resolution than just “left or right alternative”. We can even allow the developer to edit the proposal immediately.
4 Conclusions

In this paper, we have motivated why batch merge for models is not purely “an academic exercise”, and demonstrated several situations where batch merge will be of great practical use in an industrial context. We have identified a number of issues and challenges that will have to be addressed before batch merge can become a reality.

Part of the research agenda for batch merge should be revisiting existing research results from interactive model merge. Issues like operation-based [7] and state-based [10] merge will probably have to be reconsidered, just like the trade-off for Unit of Versioning and Unit of Comparison [9] will most likely change. A new model merge meta-model will be the fundamental interface between the merge engine and the merge management tools. There exist some initial ideas from various contexts [6], [3], [8] that can be used as a starting point, but there is still a long way to go.

A side effect of batch merge is a very nice and useful separation of concerns – and tools – both for industry and for research. Industry can use one tool for “merge production” and another tool(s) for “merge management” – research can proceed in parallel in both areas and will not have to do all-inclusive implementations just to try out some new research idea.

References

Comparing Model-Metamodel and Transformation-Metamodel Co-evolution

Louis M. Rose\textsuperscript{1}, Anne Etien\textsuperscript{2}, David Méndez\textsuperscript{2,3}, Dimitrios S. Kolovos\textsuperscript{1}, Richard F. Paige\textsuperscript{1} and Fiona A.C. Polack\textsuperscript{1}

\textsuperscript{1} Department of Computer Science, University of York, UK. \{louis, dkolovos, paige, fiona\}@cs.york.ac.uk
\textsuperscript{2} INRIA Lille Nord Europe, LIFL CNRS UMR 8022, Université Lille 1, France. Anne.Etien@lifl.fr
\textsuperscript{3} TICSw Research Group, Universidad de los Andes, Colombia. df.mendez73@uniandes.edu.co

Abstract. Changes to a metamodel can affect the definition and well-formedness of other development artefacts, such as models and transformations. Co-evolution, in which two or more artefacts are changed together, is one technique for managing change. This paper compares model-metamodel and transformation-metamodel co-evolution, and highlights challenges for future work on co-evolution.

1 Introduction

MDE aims to better automate engineering tasks, but introduces additional challenges for software evolution due to interdependencies between engineering artefacts. Models and transformations, for example, are interdependent with metamodels, because their definition and well-formedness depends on one or more metamodels. Interdependent artefacts complicate software evolution activities, because the evolution of one artefact may require others to be changed in response.

The interdependency between a model and its metamodel is termed conformance [1], and can be described by a set of constraints between models and metamodels [10]. Méndez et al. [9] identify a similar interdependency, domain conformance, between the definition of a transformation and its metamodels.

Co-evolution is the process of evolving several artefacts together to maintain an interdependency. This paper compares model-metamodel co-evolution and transformation-metamodel co-evolution, and synthesises challenges for their future research.

2 Existing Approaches

Existing approaches to managing model-metamodel and transformation-metamodel co-evolution can be categorised as master-slave approaches [7]. In general,
master-slave co-evolution typically involves three activities. Firstly, the master (metamodel) is changed, and in some approaches a change model (record of the evolution) is produced for use in later steps. Secondly and in an activity termed impact analysis, the extent to which dependent artefacts (models or transformations) have been affected by the metamodel evolution is assessed. Finally, change propagation is used to migrate the dependent artefacts in response to evolution.

The remainder of this section describes the way in which the latter activity, change propagation, is performed in existing model- and transformation-metamodel co-evolution approaches. Compared to change propagation, relatively little work on model-metamodel and transformation-metamodel co-evolution considers impact analysis, as discussed in Section 4.

To demonstrate the aims of the approaches discussed in this section, an example of metamodel evolution, model migration and transformation migration is made available online.

2.1 Change propagation for model-metamodel co-evolution

In [11], model-metamodel co-evolution approaches have been categorised into manual specification, operator-based and inference approaches. Each category varies in the way in the extent to which change propagation is automated. In manual specification, the metamodel developer writes by hand a migration strategy, which describes the way in which changes are to be propagated to a group of related artefacts. In operator-based approaches, the developer performs metamodel evolution by applying pre-defined operators that have a migration semantics. In inference approaches, a tool uses a change model to generate a migration strategy.

Manual Specification. Hussey and Paternostro explain the way in which the model loading mechanisms of the Eclipse Modeling Framework (EMF) [15] can be augmented with model migration code. Epsilon Flock [12] is a transformation language tailored for model migration, and also supports a diverse range of modelling technologies (such as EMF, MDR and XML).

Operator-based. COPE [5] provides a library of operators for co-evolving models and metamodels. Operators are applied to evolve a metamodel and have pre-defined migration semantics. Balancing expressiveness and understandability is a key challenge for operator-based approaches because the former implies a large number of operators while the latter a small number of operators [8, 11].

Inference. Cicchetti et al. [3] and Garcés et al. [4] use metamodel matching to infer a change model, and execute a higher-order model transformation to
determine the migration strategy. These approaches are fully automatic, but cannot infer migration strategies for every conceivable metamodel evolution.

2.2 Change propagation for transformation-metamodel co-evolution

Compared to model-metamodel co-evolution, relatively few approaches to transformation-metamodel co-evolution have been proposed. Existing transformation-metamodel co-evolution approaches use two further categories of change propagation technique.

**Event-based** Méndez et al. [9] use a metamodel monitor to report changes, and process change events to select and execute a corresponding migration strategy. User guidance is sometimes required to select an appropriate migration strategy.

**Regeneration** Roser et al. [14] deal with transformations that are generated from the bindings between an ontology and a source or target metamodel. Metamodel evolution is restricted to cases where the original and the evolved elements remain bound to the same ontological concept, and, transformations are automatically migrated by substituting the original with the evolved concept.

3 Comparison

Impact analysis and change propagation for model-metamodel co-evolution and transformation-metamodel co-evolution are now compared and contrasted.

**Impact Analysis** Impact analysis can be performed by executing conformance constraints. Failing constraints indicate model elements that have been impacted by the metamodel evolution. In this sense, impact analysis for model-metamodel and transformation-metamodel co-evolution are very similar. However, in the latter, a metamodel plays one or two roles – the source or the target – and domain conformance constraints vary between the two roles.

**Change Propagation** In [3–5, 12] model migration is expressed as a model transformation. The migrating transformation takes as input a model conforming to the original metamodel and produces as output a model conforming to the evolved metamodel. Figure 1(a) illustrates the use of a model transformation for model migration. Note that the solid vertical arrows indicate conformance, while the dashed horizontal arrows indicate domain conformance.

Similarly, transformation migration can be expressed as a model transformation. The migration transformation is higher-order\(^4\), because it consumes the

\(^4\) A model-to-model transformation that consumes or produces a model-to-model transformation is higher-order.
original transformation (domain conformant with the original metamodel) and produces the migrated transformation (domain conformant with the evolved metamodel). Figure 1(b) illustrates the use of a model transformation for transformation migration, and implies that $T_o$ is domain conformant with the original metamodel and that $T_e$ is domain conformant with the evolved metamodel.

While model and transformation migration can be expressed as a model transformation, the former is an *exogenous transformation* (the source and target metamodels differ) and the latter is an *endogenous transformation* (the source and target metamodels are the same – both are the transformation language metamodel). This difference can be seen below: in Figure 1(a) the source and target metamodels of $T_{mig}$ differ, while in Figure 1(b) they are the same.

![Diagram](image_url)  
(a) Model migration.  
(b) Transformation migration.

**Fig. 1.** Migration expressed as model transformation.

## 4 Challenges

Challenges for future co-evolution research are now synthesised.

**Supporting impact analysis** Compared to change propagation, relatively little work considers impact analysis for model- and transformation-metamodel co-evolution. Impact analysis is key to reasoning about the cost of change [16], and, hence, future research should address impact analysis for MDE.

Scalability is a key challenge for impact analysis in the context of MDE, and is considered as a fundamental requirement for the approaches described in [2, 13]. Future research should identify ways in which scalable impact analysis for MDE can be achieved, by, for example, using change models to reduce the number of constraints that are checked during impact analysis [9].

**Assessing change propagation approaches** None of the five categories of change propagation approach described in Section 2 have been applied to both model- and transformation-metamodel co-evolution. To increase understanding of their relative strengths and weaknesses, future research should continue to categorise and compare co-evolution approaches.
Common languages for expressing migration Today, a range of languages are used to specify model migration\(^5\). Identifying re-occurring patterns of model migration and building interoperable co-evolution tools are made more challenging due to this language diversity. For these reasons, future research should seek to establish a common language for expressing model migration and, likewise, for expressing transformation migration.

Unifying model-metamodel & transformation-metamodel co-evolution Given the similarities between model-metamodel and transformation-metamodel co-evolution, it may be possible to use a single tool to manage both types of co-evolution. Several potential advantages are apparent for a unified co-evolution approach. Firstly, the metamodel developer need only express evolution once to co-evolve both models and transformations. Secondly, implementation of co-evolution tools may be simplified via re-use. Thirdly, a unified approach may provide a foundation for supporting other types of co-evolution, such as model-model (more commonly termed model synchronisation). Given these advantages, further research should assess the extent to which co-evolution approaches can be unified, and compare unified and specialised approaches to establish their strengths and weaknesses.

Notwithstanding the potential advantages, a unified co-evolution approach presents some challenges. For example, the impact of a metamodel change on a model might be quite different to the impact on a transformation, and as such further research should explore the extent to which change models can be re-used in a unified co-evolution approach.

Depending on the style of change propagation employed, further challenges might exist for unified co-evolution approaches. For instance, operator-based approaches to model-metamodel co-evolution specify a library of co-evolutionary operators. Each operator specifies evolutionary information for two dimensions: metamodel evolution and model migration. Expanding the library to include a third dimension, migration semantics for transformations, will likely increase the number of operators and also present additional challenges for the usability of tools for operator application. We anticipate similar challenges for other categories of change propagation approach.

5 Conclusions

Managing evolution is key to software engineering. MDE introduces new development artefacts whose evolution poses research challenges. This paper contributes a comparison of two closely related techniques for managing evolution in the context of MDE, and outlines challenges for future research.

In our future work, we will assess transformation languages for formulating transformation migration strategies, and perform a more thorough comparison

\(^5\) The Atlas Transformation Language (ATL) [6], Groovy and Java have been used in [4, 5, 15] respectively.
of the five categories of change propagation approach for managing co-evolution. Additionally, we will explore a unification of the co-evolution approaches described in [9] and [12].

**Acknowledgement.** The work in this paper was supported by the European Commission via the MADES project, co-funded under the “Information Society Technologies” 7th Framework Programme (2009-2012), and by the Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the “Contrat de Projets Etat Region (CPER) 2007-2013.”

**References**