Models and Evolution

Joint MODELS’09 Workshop

on

Model-Driven Software Evolution (MoDSE)

and

Model Co-Evolution and Consistency Management (MCCM)
Models and Evolution

Software Evolution is a crucial, complex and very important research domain in software engineering. It has been the topic of numerous international conferences, workshops, books and scientific publications. On the other hand, it represents an omnipresent recurring problem for companies and needs practical and scalable solutions to ensure software confidence, quality and reliability. This becomes even more crucial and critical in application domains where software is geographically distributed and involves multiple stakeholders (managers, designers, developers, clients, ...) and where resources and requirements must be reconciled.

Model-driven engineering (MDE) is a software engineering methodology that focuses on models as primary software artefacts. It is meant to reduce complexity and increase productivity and reuse by raising the level of abstraction. With the advent and widespread use of MDE in academia and industry, the research topic of model-driven software evolution and co-evolution becomes more and more important. Because of this, a successful series of annual international workshops has been set up since 2007 for the MoDSE workshop and from 2008 for the MCCM workshop. Their main objective is to explore and strengthen the interaction and synergy between the active research domains of Software Evolution, Co-Evolution, Consistency Management and MDE.

Dalila Tamzalit, LINA, University of Nantes, France
Dirk Deridder, System and Software Engineering Lab, Vrije Universiteit Brussel, Belgium
Bernhard Schätz, fortiss GmbH/TU München, Germany
Program Chairs
Dalila Tamzalit, LINA, University of Nantes, France
Dirk Deridder, System and Software Engineering Lab, Vrije Universiteit Brussel, Belgium
Bernhard Schätz, fortiss GmbH/TU München, Germany

Steering Committee
Dirk Deridder, System and Software Engineering Lab, Vrije Universiteit Brussel, Belgium
Hubert Dubois, CEA LIST, France
Jeff Gray, University of Alabama, Birmingham, USA
Tom Mens, Université de Mons-Hainault, Belgium
Alfonso Pierantonio, Università degli Studi dell’Aquila, Italy
Bernhard Schätz, fortiss GmbH/TU München, 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
Benoit Baudry, IRISA, University of Rennes, France
Jorn Bettin, Sofismo, Lenzburg, Switzerland
Jean Bézivin, University of Nantes, INRIA, France
Mireille Blay-Fornarino, Université de Nice-Sophia Antipolis, France
Jean-Michel Bruel, University of Toulouse, France
Rubby Casallas, University of Los Andes, Colombia
Mirko Conrad, The MathWorks Inc., USA
Serge Demeyer, Universiteit Antwerpen, Belgium
Jean-Marie Favre, University of Grenoble, France
Reiko Heckel, University of Leicester, UK
Mehdi Jazayri, University of Lugano, Switzerland
Olivier Le Goaer, LINA, University of Nantes, France
Jason Xabier Mansell, ESI, Spain
Richard Paige, University of York, United Kingdom
Awais Rashid, University of Lancaster, United Kingdom
Jonathan Sprinkle, University of Arizona, USA
Gabriele Taentzer, University of Marburg, Germany
Antonio Vallecillo, Universidad de Malaga, Spain
Ragnhild Van Der Straeten, Vrije Universiteit, Belgium
Eelco Visser, Delft University of Technology, the Netherlands

Communication Chair
Michaël Hoste, Université de Mons-Hainault, Belgium
Co-evolution, Co-adaptation and Synchronization

An Analysis of Approaches to Model Migration, Louis Rose, Richard Paige, Dimitrios Kolovos and Fiona Polack. pages 6-15

A Strategy for Synchronizing and Updating Models after Source Code Changes in Model-Driven Development, Rafael Meneses and Rubby Casallas. pages 16-26

Inconsistency Analysis at Integration of Evolving Domain Specific Models based on OWL, Christian Bartelt. pages 27-37

Tooling and Industrial feedback

Model Refactoring in Eclipse by LTK, EWL, and EMF Refactor: A Case Study, Thorsten Arendt, Florian Mantz, Lars Schneider and Gabriele Taentzer. pages 38-47

Implementing An Inconsistency Management Framework for Model Transformation Tools, Olaf Muliawan and Dirk Janssens. pages 48-57

Comprehensive Support for Evolving Software Cases, Michal Smialek. pages 58-67

Practical case study of MDD infusion in a SME: Final Results, Regis Vogel. pages 68-78

Why Model Versioning Research is Needed!? An Experience Report, Kerstin Altmanninger, Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland and Manuel Wimmer. pages 79-90

Modelling Evolution

Limitations of Automating Model Migration in Response to Metamodel Adaptation, Markus Herrmannsdoerfer and Daniel Ratiu. pages 91-100

Model Patches in Model-Driven Engineering, Antonio Cicchetti, Davide Di Ruscio and Alfonso Pierantonio. pages 101-110

Change Impact Analysis of Model-Driven Development Systems using Evolution Scenario Templates, Bedir Tekinerdogan and Ersin Er. pages 111-121

Evolution Techniques

Service Oriented Architecture Definition Using Composition of Evolutions, Sebastien Mosser, Mireille Blay-Fornarino and Michel Riveill. pages 122-131
State-based vs. Operation-based Change Tracking, Maximilian Koegel, Markus Herrmannsdoerfer, Jonas Helming and Yang Li. pages 132-141

Using change intentions to guide evolution and versioning in Model Driven Development, Rafael Leaño, Rubby Casallas and Nicolás López. pages 142-151
An Analysis of Approaches to Model Migration

Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos and Fiona A.C. Polack

Department of Computer Science, University of York, UK.
[louis,paige,dkolovos,fiona]@cs.york.ac.uk

Abstract. Changing a metamodel can cause inconsistency with instance models. Modelling frameworks such as EMF that implement the OMG's metamodelling architecture cannot be used to manipulate models that are inconsistent with their metamodel: inconsistent models cannot be loaded. Consequently, developers have to invest effort in migrating inconsistent models, to re-establish consistency. This is an example of model and metamodel evolution – indeed, of co-evolution. In their recent book, Mens and Demeyer note that there are new challenges for MDE in controlling and managing model and metamodel evolution. Various approaches to automating model migration have been proposed. In this paper, we demonstrate some of the important limitations of automatic generation of model migration strategies, and identify requirements for improved model and metamodel co-evolution support.

1 Introduction

Software development often involves constructing a system by combining numerous types of interdependent artefact (such as source and object code, build scripts, documentation and configuration settings). Some examples of these dependencies include: compiling object code from source code, generating documentation from source code, and deploying object code using a build script. When an artefact is changed, it may become inconsistent with other development artefacts.

The definition of consistency varies depending on the artefacts being considered. For example, Java object and source code might be considered consistent when the object code contains a .class file for each .java file in the source code. This paper focuses on model and metamodel consistency, which is discussed below.

Co-evolution is a development activity in which several artefacts are changed together to maintain consistency. Co-evolution can be performed by first changing a single artefact (e.g. source code). Then each dependent artefact is changed to re-establish consistency; we term this activity migration. Some types of migration are often automated (e.g. background incremental compilation of source code to object code), while some must be performed manually (e.g. updating design documents after adding a new feature).
1.1 Model and Metamodel Co-Evolution

Model-Driven Engineering (MDE) introduces additional challenges for controlling and managing evolution [15]. For example, when a metamodel is changed, its models may need to be migrated so as to re-establish model and metamodel consistency.

A model and metamodel are consistent when the model conforms [2] to the metamodel (i.e. when the metamodel specifies every concept used in the model definition, and the model uses the metamodel concepts according to the rules specified by the metamodel). Conformance can be described by a set of constraints between models and metamodels [17]. (For example, a model and metamodel conformance constraint might state that every object in the model has a corresponding non-abstract class in the metamodel). When all constraints are satisfied, the model is said to conform to the metamodel.

In some modern MDE environments – such as the popular Eclipse Modelling Framework (EMF) – models and metamodels are normally kept separate; in Eclipse, instances of a metamodel reside in a different workspace to the metamodel. Because of this, co-evolution is typically a two-step process. First, the metamodel is evolved and then the instance models are migrated. During metamodel evolution, a migration strategy is devised and then distributed to the workspace containing instance models. In the workspace containing the instance models, the migration strategy is executed to migrate the instance models.

Several automatic and semi-automatic approaches for generating model migration strategies have been proposed [4, 8, 10, 20, 21]. Each approach aims to reduce the effort required to perform model and metamodel co-evolution, but has some limitations. In Section 2, existing approaches for generating model migration strategies are categorised and analysed. Alternative approaches for managing model and metamodel co-evolution are described in Section 3. Finally, Section 4 summarises the analysis contributed in Section 2 and proposes future work.

2 Existing Approaches to Model Migration

This section contributes a categorisation of existing approaches to model migration, and discusses the relative merits and limitations of each category. When performing metamodel evolution, a metamodel developer can apply any approach discussed in this section to generate a corresponding migration strategy, which can then be executed to perform model migration.

2.1 Manual Specification

In manual specification, the migration strategy is encoded manually by the metamodel developer, typically using a general purpose programming language (e.g. Java) or a model-to-model transformation language (such as QVT [16], or ATL [12]). The migration strategy can manipulate instances of the metamodel in
anyway permitted by the modelling framework and hence manual specification permits the metamodel developer the most control over model migration.

However, manual specification generally requires the most effort on the part of the metamodel developer for two reasons. Firstly, as well as implementing the migration strategy, the metamodel developer must also produce code for executing the migration strategy. Typically, this involves integration of the migration strategy with the modelling framework (to load and store models) and possibly with development tools (to provide a user interface). Secondly, our experiences with specifying model migration strategies has indicated that several patterns re-occur frequently, such as copying a model element from original to migrated model. These patterns are not captured by existing general purpose and model-to-model transformation languages, and so each metamodel developer has to devise his or her own abstractions.

2.2 Operator-based Co-evolution

In operator-based co-evolution, a library of co-evolutionary operators is provided. Each co-evolutionary operator specifies a metamodel evolution along with a corresponding model migration strategy. For example, the “Make Reference Containment” operator evolves the metamodel such that a non-containment reference becomes a containment reference and migrates models such that the values of the evolved reference are replaced by copies. By composing co-evolutionary operators, metamodel evolution can be performed and a migration strategy can be generated without writing any code. Wachsmuth [21] proposes a library of co-evolutionary operators for MOF metamodels. COPE [10] is an operator-based co-evolution approach for the Eclipse Modeling Framework.

The usefulness of an operator-based co-evolution approach depends heavily on the richness of the library of co-evolutionary operators that it provides. When no appropriate co-evolutionary operator is available, the metamodel developer must use another approach for performing model migration. COPE allows metamodel developers to specify custom migration strategies when no co-evolutionary operator is appropriate, using a general purpose programming language. Consequently, custom migration strategies in COPE suffer one of the same limitations as manual specification approaches: model migration patterns are not captured in the language used to specify migration strategies.

Therefore, it seems that operator-based co-evolution approaches should seek to provide a complete library of co-evolutionary operators, so that at least one operator is appropriate for every co-evolution that a metamodel developer may wish to perform. However, as discussed by Lerner [14], a large library of operators increases the complexity of specifying migration. To demonstrate, Lerner considers moving a feature from one type to another. This could be expressed by sequential application of two operators called, for example, delete_feature and add_feature. However, the semantics of a delete_feature operator likely dictate that the values of that feature will be removed during migration and hence, delete_feature is unsuitable when specifying that a feature has
been moved. To solve this problem, a move_feature operator could be introduced, but then the metamodel developer must understand the difference between the two ways in which moving a type can be achieved, and carefully select the correct one. Lerner provides other examples which further elucidate this issue (such as introducing a new type by splitting an existing type). As the size of the library of co-evolutionary operators grows, so does the complexity of selecting appropriate operators and, hence, the complexity of performing metamodel evolution.

Clear communication of the effects of each co-evolutionary operator (on both the metamodel and its instance models) can improve the navigability of large libraries of co-evolutionary operators. COPE, for example, provides a name, description, list of parameters and applicability constraints for each co-evolutionary operator. An example, taken from\(^1\), is shown below.

**Make Reference Containment**

In the metamodel, a reference is made containment. In the model, its values are replaced by copies.

*Parameters:*
- reference: The reference

*Constraints:*
- The reference must not already be containment.

Other techniques can be used to try to improve the navigability of large libraries of co-evolutionary operators. For example, COPE restricts the choice of operators to only those that can be applied to the currently selected metamodel element. Finding a balance between a richness and navigability is a key challenge in defining libraries of co-evolutionary operators for operation-based co-evolution approaches. Analogously, a known challenge in the design of software interfaces is the trade-off between a rich and a concise interface \([3]\).

To perform metamodel evolution using co-evolutionary operators, the library of co-evolutionary operators must be integrated with tools for editing metamodels. COPE, for instance, provides integration with the EMF tree-based metamodel editor. However, some developers edit their metamodels using a textual syntax, such as Emfatic \([11]\). In general, freeform text editing is less restrictive than tree-based editing (because in the latter, the metamodel is always structurally sound whereas in the former, the text does not always have to compile). Consequently, it is not clear whether operator-based co-evolution can be used with all categories of metamodel editing tool.

### 2.3 Metamodel Matching

In *metamodel matching*, a migration strategy is inferred by analysing the evolved metamodel and the *metamodel history*. Metamodel matching approaches use one of two categories of metamodel history; either the original metamodel (*differencing* approaches) or the changes made to the original metamodel to produce the

evolved metamodel (change recording approaches). The analysis of the evolved metamodel and the metamodel history yields a difference model [4], a representation of the changes between original and evolved metamodel. The difference model is used to infer a migration strategy, typically by using a higher-order model-to-model transformation\textsuperscript{2} to produce a model-to-model transformation from the difference model. Cicchetti et al. [4] and Garcés et al. [8] describe metamodel matching approaches. More specifically, both describe differencing approaches. We are not aware of any existing change recording approaches, although COPE [10] uses change recording to support the specification of custom model migration strategies.

Compared to manual specification and operator-based co-evolution, metamodel matching requires the least amount of effort from the metamodel developer who needs only to evolve the metamodel and provide a metamodel history. However, for some types of metamodel change, there is more than one feasible model migration strategy. For example, when a metaclass is deleted, one feasible migration strategy is to delete all instances of the deleted metaclass. Alternatively, the type of each instance of the deleted metaclass could be changed to another metaclass that specifies equivalent structural features.

In general, a metamodel matching approach requires guidance to select the most appropriate migration strategy from all feasible alternatives; the metamodel changes alone do not provide enough information to correctly distinguish between feasible migration strategies. Existing metamodel matching approaches use heuristics to determine the most appropriate migration strategy. These heuristics sometimes lead to the selection of the wrong migration strategy. An alternative to using heuristics would be to present the metamodel developer with all feasible migration strategies, allowing him or her to select the most appropriate one.

Because metamodel matching approaches use heuristics to select a migration strategy, it can sometimes be difficult to reason about which migration strategy will be selected. For domains where predictability, completeness and correctness are a primary concern (e.g., safety critical or security critical systems, or systems that must undergo certification with respect to a relevant standard), such approaches are unsuitable, and deterministic approaches that can be demonstrated to produce correct, predictable results will be required.

Before discussing the benefits and limitations of differencing and change recording metamodel matching approaches, we introduce an example of co-evolution.

**Example** The following example was observed during the development of the Epsilon FPTC tool (described in [18]). Figure 1 illustrates the original metamodel in which a System comprises any number of Blocks. A Block has a name, and any number of Successor Blocks; predecessors is the opposite of the successors reference.

\textsuperscript{2} A model-to-model transformation that consumes or produces a model-to-model transformation is termed a higher-order model transformation.
Further analysis of the domain revealed that extra information about the relationship between Blocks needed to be stored. The evolved metamodel is shown in Figure 2. The Connection class is introduced to capture this extra information. Blocks are no longer related directly to Blocks, instead they are related via an instance of the Connection class. The incomingConnections and outgoingConnections references of Block are used to relate Blocks to each other via an instance of Connection.

A model that conforms to the original metamodel (Figure 1) might not conform to the evolved metamodel (Figure 2). Below is a description of the strategy used by the Epsilon FPTC tool to migrate a model from original to evolved metamodel:

1. For every instance, b, of Block:
   - For every successor, s, of b:
     -- Create a new instance, c, of Connection.
     -- Set b as the source of c.
     -- Set s as the target of c.
     -- Add c to the connections reference of the System containing b.
2. And nothing else changes.

Differencing and change recording metamodel matching approaches are now compared and contrasted.
Change recording In change recording approaches, metamodel evolution is monitored by a tool, which records a list of primitive changes (e.g. Add class named Connection, or Change the type of feature successors from Block to Connection). The record of changes may be reduced to a normal form, which might remove redundancy, but might also erase useful information. In change recording, some types of metamodel evolution can be more easily recognised than with differencing. With change recording, renaming of a metamodel element from X to Y can be distinguished from the following sequence: remove a metamodel element called X, add a metamodel element called Y. With differencing, this distinction is not possible.

In general, more than one combination of primitive changes can be used to achieve the same metamodel evolution. However, when recording changes, the way in which a metamodel is evolved affects the inference of migration strategy. In the example presented above, the outgoingConnections reference (shown in Figure 2) could have been produced by changing the name and type of the successors reference (shown in Figure 1). In this case, the record of changes would indicate that the new outgoingConnections reference is an evolution of the successors reference, and consequently inferred migration strategy would likely migrate values of successors to values of outgoingConnections. Alternatively, the metamodel developer may have elected to delete the successors reference and then create the outgoingConnections reference afresh. In this record of changes, it is less obvious that the migration strategy should attempt to migrate values of successors to values of outgoingConnections. Change recording approaches require the metamodel developer to consider the way in which their metamodel changes will be interpreted.

Change recording approaches require facilities for monitoring metamodel changes from the metamodel editing tool, and from the underlying modelling framework. As with operation-based co-evolution, it is not clear to what extent change recording can be supported when a textual syntax is used to evolve a metamodel. A further challenge is that the granularity of the metamodel changes that can be monitored influences the inference of the migration strategy, but this granularity is likely to be limited by the modelling framework.

Differencing In differencing approaches, the original and evolved metamodels are compared to produce the difference model. Unlike change recording, metamodel evolution may be performed using any metamodel editor; there is no need to monitor the primitive changes made to perform the metamodel evolution. However, as discussed above, not recording the primitive changes can cause some categories of change to become indistinguishable, such as renaming versus a deletion followed by an addition.

To illustrate this problem further, consider again the metamodel evolution described above. A comparison of the original (Figure 1) and evolved (Figure 2) metamodels shows that the references named successors and predecessors no longer exist on Block. However, two other references, named outgoingConnections and incomingConnections, are now present on Block. A meta-
model matching approach might deduce (correctly, in this case) that the two new references are evolutions of the old references. However, no metamodel matching approach is able to determine which mapping is correct from the following two possibilities:

- successors evolved to incomingConnections, and predecessors evolved to outgoingConnections.
- successors evolved to outgoingConnections, and predecessors evolved to incomingConnections.

The choice between these two possibilities can only be made by the metamodel developer, who knows that successors (predecessors) is semantically equivalent to outgoingConnections (incomingConnections). As shown by this example, fully automatic differencing approaches cannot always infer a migration strategy that will capture the semantics desired by the metamodel developer.

3 Related Work

Although existing model and metamodel co-evolution approaches focus mostly on automatic or semi-automatic generation of migration strategies, other approaches exist. EPatch [7] is a language for describing model differences, similar to the UNIX patch language. The EPatch tools, part of EMF Compare, can be used to compare metamodels and derive an EPatch description. Metapatch [7] is used for specifying model migration strategies by hand (manual specification). Metapatch model migration strategies are embedded in EPatch metamodel change descriptions.

In some cases, such as when metamodel evolution has caused only a small number of model and metamodel inconsistencies, manual migration of instance models may be less time consuming than specifying an executable migration strategy (by using one of the approaches discussed in Section 2). In [19], we describe a semi-automatic approach for managing model and metamodel inconsistency, which is well suited to the reconciliation of small numbers of model and metamodel inconsistencies.

Related work can also be found in the fields of database and XML schema evolution. Many existing approaches to managing schema evolution, such as [1, 6, 9, 13], share a similar method to the operator-based co-evolution approaches described in Section 2.2 – they define a library of evolutionary operators – and, as such, suffer from many of the same challenges and limitations. As discussed in Section 2.2, Lerner [14] notes that balancing the size and navigability of a library of evolutionary operators is a key challenge for these approaches. In addition, Lerner proposes an alternative approach, analogous to the metamodel matching approaches discussed in Section 2.3, in which original and evolved versions of a schema are compared to infer a migration strategy. In [5], de Vries and Roddick propose the introduction of an extra architectural layer to a typical relational database management system. The extra layer – which contains mathematical
relations (termed mesodata) for describing the way in which old values should be migrated to new values – can be used to perform migration in response to the type of an attribute changing.

4 Conclusions and Further Work

This paper has described existing approaches for managing model and metamodel co-evolution, and contributed an analysis of approaches that automatically or semi-automatically generate migration strategies. A categorisation of the analysed approaches was presented, and the benefits and limitations of each category of approach were identified.

In summary, manual specification of migration strategies affords the metamodel developer the most control over the way in which instance models will be migrated, but requires more effort from the metamodel developer than semi-automatic and automatic approaches. Operator-based co-evolution provides the metamodel developer with a library of co-evolutionary operators for performing metamodel evolution. A migration strategy can be inferred by composing the co-evolutionary operators chosen by the metamodel developer. In situations where no co-evolutionary operator is suitable, the metamodel developer must use another approach to generate a migration strategy. The usefulness of an operator-based co-evolution approach is heavily influenced by both the richness and the navigability of its library of co-evolutionary operators. Generating a migration strategy with a metamodel matching approach requires the least effort from the metamodel developer. However, in general metamodel matching approaches cannot be guaranteed to infer a migration strategy that captures the semantics desired by the metamodel developer, as shown by the example in Section 2.3. As such, metamodel matching approaches use heuristics to decide between feasible migration strategies, but, in domains where predictability, completeness and correctness are a primary concern, deterministic approaches that can be demonstrated to produce correct, predictable results are required.

In future work, we plan to address some of the limitations and challenges summarised above. In particular, we envisage devising improved mechanisms for communicating the effects of co-evolutionary operators. We will also continue to investigate patterns that frequently occur when general purpose and model-to-model transformation languages are used to describe model migration.

Acknowledgement. The work in this paper was supported by the European Commission via the MODELPLEX project, co-funded by the European Commission under the “Information Society Technologies” Sixth Framework Programme (2006-2009).

References


A Strategy for Synchronizing and Updating Models after Source Code Changes in Model-Driven Development

Rafael Meneses and Rubby Casallas

Grupo de Construcción de Software, Universidad de los Andes, Bogotá, Colombia

{rg.meneses81,r.casallas}@uniandes.edu.co

Abstract. In this paper, we propose a model-based synchronization strategy to facilitate the updating of models according to source code changes into a model-driven approach. In the context of model-driven development, the source code generation is not complete; therefore, developers have to modify the generated source code to achieve entirely the system requirements. This evolution process causes a loss of synchronization between software models and code. Due to the lack of tools to support the evolution changes, the synchronization process is carried out manually. Our work focuses primarily on maintaining the software models and code representation of a system synchronized, facilitating the automatic propagation of changes made on the source code to the models from which it was generated. Our proposal is based on different model-driven approaches, such as AST model discovery, model traceability techniques, model-to-model transformations and model matching strategies.

Keywords: Model-Driven Engineering, Model-Code Synchronization, Software Product Lines, Traceability, Model Transformations, Model Comparison.

1 Introduction

Model Driven Software Engineering (MDE) has as main objective using high abstraction level models throughout the whole development lifecycle [1]. The basic premise of this development paradigm is to focus the development and maintenance efforts in manipulating models as opposed to source code.

Software systems need to evolve, and MDE systems are not the exception [2]. In MDE, high level software artifacts are represented in models (business processes, architecture, platform, design, etc); based on these models an MDE framework aims at generating the source code for the actual running system. However it is not always possible to generate 100% of the source code. As a consequence, developers must manually modify some implementation artifacts in order to complete the requirements of the system. However, this can lead to a loss of synchronization between the high level models and the generated code.
In this paper we present a strategy to maintain synchronization between models and code that allows for coherent evolution of the models according to some changes that the developers make manually on the generated code. Our proposal asks the developer the parts of code written manually which must have an impact on the higher level models; with this information we automatically update the models that represent the domain problem. In this manner, changes that the developers introduce manually on implementation artifacts can be included as part of the generated code afterwards.

This paper is organized as follows: Section 2 details the synchronization problem and presents the motivation that drives our solution. Section 3 presents our strategy to solve this issue and introduces an illustrative example. Section 4 compares our work with others, and finally, Section 5 concludes and sketches out some of our future works.

2 Problem: Maintaining synchronization between code artifacts and models

Our focus is Model Driven Software Product Lines (MD-SPLs). An MD-SPL uses as main assets metamodels, models and transformations to generate concrete products in the line [3]. In this context, models describe specific details of a product in the family by instantiating concepts of a specific domain (e.g., business, architecture, platform, language, etc.). These domains separate concerns; so for example, in the business domain the analyst must model business concepts, without taking into consideration the specific platform technology or architectural style. These lower level details are part of the lower level domains of the product line.

In an MD-SPL, we need to execute successive transformations to close the gap between the problem and solution domains. These transformations add details into lower level models, until finally, to obtain the executable source code for the application. However, implementing transformations that generate 100% of the source code for an application is very difficult.

For this reason, the developers must manually modify the source code generated in order to complete the business requirements. In a scenario where a product in the line must evolve, usually, developers make most of the changes on the implementation artifacts. Maintaining and evolving the models and regenerating code are activities no longer viable since the new version of the code does not include the changes the developers made manually. As a result, the model and the code become incoherent: models represent the state of the system during the design phase, but the source code represents the current implementation state. This is called a loss of synchronization.

To avoid this problem and ensure the consistent evolution between models and code, the models should be updated based on manual changes made over the code. It should be noticed that not all changes must be included in the updating process, because there are many manual modifications only related to the implementation artifacts, which do not need to be represented at model level (e.g. a statement in a method).
A model-driven development process is composed of different and several domains. In this context, to deal with the problem of semi-automatic updating of models, is necessary to traverse in reverse direction the transformation chain in order to reach the highest level model. This model represents the problem domain and it is the starting point for carrying out the regeneration process of the source code.

Our synchronization proposal allows identifying sections of source code written by developers to be included in the artifacts of high level and uses this information to: (a) achieve the highest level model representing the problem through different domains that make up the MD-SPL and (b) update that model so that the information of the changes is included in a code regeneration process.

The proposed solution uses model-based techniques such as the AST model discovery from source code and model matching strategies to find the code written manually by developers on the generated code. Likewise, uses model-to-model transformations and model traceability techniques to obtain the information related to the different domains of a MDE system from the source code artifacts.

Next section presents a detailed review of the proposed solution strategy.

3 Our Strategy

3.1 Case Study: Model Refinement Line for Enterprise Applications

Our proposed strategy is based on using The Model Refinement Line for Enterprise Applications (MRL-EA) that we created in previous projects. The main objective of the MRL-EA is to provide an MDA based, agile, semiautomatic process for development of multilayered applications using JEE5 technology. Products created in the Line offer basic CRUD (create, read, update, delete) functionalities for business entities.


Figure 1 shows the different domains and the four refinement steps that make up the Line. The Line defines a high level model, which represents the business model of the application. This model is built by a business designer using concepts that represent the domain problem. The first transformation converts this business model into an architecture model. This architecture model contains all of the concepts that describe a multilayered application.

The next steps (platform and language) produce, respectively, the design and implementation models for the system. The second transformation includes in the architecture model concepts specifically related to JEE5 platform. The third transformation refines this result by generating a model that defines the business application in terms of Java specific concepts. Finally, in the last step, the developer can use a model-to-text transformation to create executable code in Java.
In the MRL-EA the code generation process receives as input a business model and automatically builds the refined models in the other domains, as well as the source code for the application.

### 3.2 Proposed solution

In the context of the MRL-EA, to maintain synchronization between models and generated source code, we must trace the changes identified in the source code all the way up to the business model. This is because the business model is the main input to start the generation process. In this sense, it would be useless to make changes in one of the intermediate models in the line, since these models are overwritten each time we execute the model-to-model transformations.

The actual source code produced by the MRL-EA is a result of higher level models and contains information from each domain in the line because of the successive transformations executed for its generation. Thus, to update the business model according to modifications introduced manually by the developer in the source code, we need to trace back the refinement chain in the opposite direction and recover domain information for each of the successive previous domains until we reach the initial model. With this strategy we can identify the parts of the business model where we must make changes that correspond to the changes introduced manually by the developer.

The developer can modify the generated source code in many different ways, for example, he can add, modify or eliminate parts of the code, including attributes, classes and relationships between classes. Our main challenge, additionally to identifying the nature of the change, is to understand its intention which determines the artifact which is the object of the change (i.e., the actual concept from the business model that the developer wants to impact). For example, when we identify that the developer has added a new attribute to a class, we need to distinguish if his intention is to include a new characteristic to the respective model entity or if he just needs to create a new attribute to store an intermediate value.

The changes made manually to the source code that we must trace to the domain problem must be directly related to information that can be represented in the business...
metamodel. It is not possible to trace everything, specifically information that does not correspond to concepts of the business domain.

Our model synchronization approach allows the developer to identify the changes made manually over the source code and identify the explicit intention of the specific changes that must be traced to the domain problem. With this information, we go backwards in the model transformations to identify the specific components of the business model that we must update. Finally using the information recollected in this process, we update automatically the business domain model.

In the following section we detail the steps and the algorithm of the solution we implemented and describe briefly the model based tools we use in each step.

3.3 Synchronization process

The synchronization process includes the following steps:

**Code-to-model Transformation:** We use ASTs (Abstract Syntax Trees) to model the source code. An AST model represents the syntax tree of a piece of code; each element in the tree represents a construct of the source code [7].

Our strategy to identify changes made manually to the source code is to find the differences between the code generated by the MRL-EA and the modified code. To achieve this, the first step is to generate automatically two AST models. The first one represents the original source code; this AST model is built each time we generate code using the MRL-EA. The second one is built each time we execute the synchronization process to represent the current state of implementation artifacts.

To do this task we use a generic tool of the MoDisco (Model Discovery) project [8] called *Java Abstract Syntax Discovery Tool*[^1]. This tool allows the construction of models that represent the abstract syntax tree of source code in Java.

**Comparison of AST models:** In this activity we compare the AST models produced in the previous step and as result we identify the specific changes made manually on the source code. It is important to note that although our approach allows the developer to identify some manual changes explicitly, in some cases we need to automatically identify all changes made by hand. For example, to validate that the intention of the change specified by the developer matches the code associated with that intention. For this reason, this comparison process allows to find and categorize any changes between two different versions of source code.

The comparison process uses a matching strategy to find model differences implemented using a DSL called AML (AtlanMod Matching Language) proposed by Kelly Garces in [9]. This strategy allows us to identify changes and categorize them broadly as a change that adds, eliminates or renames an attribute, method or class. Since a single class in Java cannot have two attributes with the same name, we use the attribute name to discover changes. To detect changes in methods, the strategy trusts

[^1]: Tool provided by INRIA AtlanMod group. Its development is supported by IST European MODELPLEX project (MODELing solution for comPLEX software systems, FP6-IP 34081).
in the name of the method, the return type and the parameters. Changes in the classes could be identified by the name of the class, as in the attributes and methods it contains.

Using AML to compare the AST models we produce a weaving model that defines the differences in the two models. Each difference references elements of (sometimes) both models that are included in the change. For example, if the difference is the addition of an attribute, the element that represents this difference references only the attribute in the AST model for the modified source code since in the AST model for the generated code there is no correspondence to this attribute. In the case of the elimination of an attribute, the weaving model only references the attribute in the AST model for the generated code. The weaving model can be represented visually using the AMW (AtlanMod Model Weaver) tool [10].

Trace model creation. Once we identify the specific changes between the generated and modified versions of the source code, we must use this information to locate the specific components in the business model that are impacted by each change. To achieve this, we trace information of the changes backwards through the transformations of the MRL-EA.

Each change identified in the weaving model had to be introduced manually in a specific source code artifact (e.g., an attribute added must be included inside of a class). Each of these artifacts can be indirectly related to an artifact in the initial model corresponding to a business concept. Thus, using the trace information, we can identify the element of the initial model from which a specific artifact in the source code was generated.

This trace information refers to the relationships between each source element (in each source model of the MRL-EA) and its corresponding objective models (the results of each transformation of the MRL-EA). To obtain this information we use a mechanism for explicit trace information access in ATL proposed by Andrés Yie in [11]. This mechanism allows us to store trace information automatically in a traceability model each time we execute the transformation rules. This strategy, besides making trace information explicit, does not require modifications to the code of each transformation.

Using the trace mechanism, each time we execute a generative process, the MRL-EA produces, additionally to the destination models, traceability models corresponding to each of the three transformations. Each of these models characterizes the relationships between a source and an objective model.

The final activity of this step is merging the traceability models in a single model that contains direct relationships between the artifacts of the source code and the elements of the business model. With this single model we can identify the specific elements of the business model that must be modified according to the differences in the AST models.

Reconciliation: We need to have some information regarding each change to update the initial model of the MRL-EA which varies for each type of change identified (adding, eliminating or renaming an element). Let’s return to our example where a developer adds a new attribute in the source code. To update the business model according to this change we need to: (a) understand its intention (in this case, adding
an attribute to the business model), this is relevant because not all additions of an attribute must be traced to the initial model; (b) obtain the name, type, and some additional properties of the element changed (e.g., the values of obligatoriness and uniqueness of an attribute in the MRL-EA); and (c) identify the specific name of the entity to be modified in the initial model of the MRL-EA from the Java class that contains the manual change.

These data can be obtained from different sources. We obtain the information regarding the elements of the initial model that are modified from the traceability model. We obtain the type and some additional information regarding the specific change (e.g., the name and type of the attribute added) from the weaving model and its associations with the AST models.

However, the specific intention of the change must be explicitly indicated by the developer for us to be able to decide if the modification must be synchronized with the business model. A developer can include many new attributes manually in the source code, however, only some of them must be traced to the initial model. We have defined specific code annotations to allow the developer to specify this intentionality. The developer can use these annotations on a new attribute in the following way: `@AddBusinessAttribute(required="true", unique="true")`. In this specific case, the annotation has two parameters that specify additional information necessary to modify the business model which cannot be obtained from the source code or any of the models in the line.

Based on the work about first-class change objects presented by Peter Ebraert in [12], we have built a change metamodel that contains the necessary concepts to represent the information required to execute the reconciliation of the domain model and the changed source code (Figure 2). We use the AST models, the weaving model and the general trace model to build a model that corresponds to the change metamodel; this process is called the Reconciliation process. The model produced in this process, besides allowing us to represent clearly in a single artifact all of the information necessary to modify the initial model, eases the update process of the initial model.

Figure 2 presents the change metamodel. This metamodel allows for modeling of changes of different types as well as properties (Property) for each type of change. The Subject concept refers to the target element of the initial model on which the change will have an impact.

![Change Metamodel](image)
Figure 3 (LHS) presents an example where a new attribute (address) is added (and annotated) in a source code class (UniversityBO) and its representation in a change model (right side of Figure 3).

![Figure 3](image)

**Fig. 3.** Example of a change that adds an attribute and its representation in a change model

In the model on the right side of figure 3, the association between the change element add and the subject element Attribute means that the change corresponds to the addition of an attribute. Each of the properties of the model corresponds to a datum necessary and mandatory to realize the respective change in the initial model of the line. The properties Type and Name specify the data type and name of the attribute that must be added, these two values are obtained from the change identified in the difference model and its relation with the AST model of the modified source code.

The values of properties Unique and Required are obtained directly from the attributes of the @AddBusinessAttribute annotation, which was manually introduced by the developer, and are represented in the AST model of the modified source code. The values for the last two properties in the model, EntityType and EntityName are obtained from the general traceability model. In this case, the values of these properties, BusinessEntity and University respectively, refer to the type and name of the business entity in the high level model from which the class UniversityBO was generated where the change is contained.

**Model Update:** The last step of our strategy is updating the business model based on the change model which was produced in the Reconciliation step. We implemented an ATL transformation for this purpose that receives as input the original business model and the change model. The transformation adds, modifies or eliminates the necessary elements according to the change model and, additionally, copies all of the elements of the original model that remain unchanged.

Continuing with our example where the developer adds a new attribute (Figure 3), Figure 4 presents the entity University of the business model, before (LHS) and after (RHS) the execution of the update process. The attribute name remains unchanged and the attribute address, which was added to the source code, now is one of the attributes of the University entity.
With respect to the maturity of the project, our current implementation covers all steps of the strategy outlined above. However, only allows automatically add attributes and methods from changes identified in the source code. Our current work is based on completing this implementation to support more complex changes, such as adding new classes, along with their respective associations and the eliminate and renaming of attributes, methods and classes.

4 Related Work

Several methods and tools have been developed based on Reverse Engineering and Round-trip Engineering processes to achieve reconcile artifacts of implementation stage with the artifacts of the design and analysis stages. In this section, we focus on the related works closer to MDE.

In [13] the most widely used CASE tools to generate UML models from the source code of applications are exposed. This is the case of Rational XDE, Borland Together, Eclipse GMT and Fujaba (From UML to Java and Back Again). Some, like Borland LiveSource, allows for synchronization of implementation models with the system source code. Fujaba for instance, creates class diagrams and history diagrams (combination of activity and collaboration UML diagrams) beginning with the abstract syntax tree extracted from the source code, which can be modified manually using special naming conventions.

These tools do not allow for creation of diagrams different from the ones referent to the classes (except from Fujaba), only allow for generation, and in some cases, synchronization of UML models with very low level of abstraction. In a MDE context, characterized by different levels of abstraction represented by customized models and metamodels, it is necessary the source code elements correspondence with the higher level models and not only with the models that represent the status of the implementation artifacts.

In [14], Czarnecki and Antkiewicz propose an agile round-trip engineering approach inspired by the Concurrent Versioning System (CVS). This strategy updates independently the models and the source code and keeps these artifacts synced on demand. The context in which this proposal is made corresponds to the Framework-Specific Modeling Languages (FSMLs), where models are constructed from abstractions of a certain framework and are used to generate the source code on that framework. In our context, this is a problem because this solution is designed for the
modeling of a particular problem and some troubles can occur (as the migration of an application that is based on a specific framework to a different framework). In addition, this approach does not propose any strategy to automatically take decisions for reconciliation between model and code and therefore it must be done manually.

Another model-code synchronization technique based on Round-trip Engineering is presented in [7]. Similar to the Czarnecki’s proposal, it allows for independent updating of artifacts to be synchronized. As our strategy, it uses the AST representation of the original code and the manually modified code. The reconciliation between the artifacts from high and low level is done automatically in main memory through a three-way tree merging process among both AST representations.

This technique, although identifying and synchronizing changes in the source code with instruction-level granularity, does not propose any solution to the problem of synchronization among the code and the higher level models that represent the problem domain. This work is based on calculating the differences at code level and updating the model that represents the lowest level domain in a MDE system. This leads to an additional disadvantage, because it makes difficult to identify the intention of a modification made by a developer on the code.

5 Conclusions and Future Work

The model-code synchronization problem presents several challenges associated with working with customized higher level models and the conceptual gap that has always existed between such artifacts and the implementation elements within the software life cycle. In this paper, we present a synchronization strategy to maintain updated the highest level models of a MDE system, based on the manual changes made in the source code generated by the successive transformations of this system.

Our proposal allows relatively closing the gap between the problem space (represented by the models of the highest level) and the solution space (represented by the source code), thanks to the trail in the opposite direction that makes it through the different domains that compose the system. In the same way, it allows to identify the intention of some modifications made by the developers, and thus determine which of them should be transferred to the initial models.

The contribution of our proposal is to reduce the model-code synchronization problem to a synchronization problem between models of different domains. Through AST models it is possible to manipulate indirectly the source code and its inclusion in model-to-model transformations.

The current implementation of our strategy includes all the steps described in the Section 3.3. This implementation enables the identification of manual changes of addition of attributes and methods in a class of source code and its subsequent inclusion in entities of the initial model of the refinement line. As future work we plan to finish the current implementation to support addition changes of full classes, and the elimination and renaming of attributes, methods and classes. In addition, we plan to design and execute extensive tests that allow us to validate each of the steps in the strategy and if necessary make adjustments to the technologies used.
References

Inconsistency Analysis at Integration of Evolving Domain Specific Models based on OWL

Christian Bartelt

Clausthal University of Technology – Software Systems Engineering Group, Adolph-Roemer-Straße 2A, 38678 Clausthal-Zellerfeld christian.bartelt@tu-clausthal.de

Abstract. Model driven engineering is one answer to increasing demands on software development and maintenance. Today’s software systems are often large, complex but also safety-critical and should be highly adaptable in life cycle. The efficient development of large and complex software systems needs a high degree of collaboration in the design and specification phases. Well-defined, (graphical) modeling languages provide therefore a matter of communication for software engineers. Besides the distribution of development locations, the co-evolution of models and the concurrency of work are typical in global software engineering projects. This kind of collaborative modeling needs reliable integration mechanisms for co-evolved model artifacts. However, the syntactic and semantic correct (consistent) integration of concurrent evolved models is not satisfactorily supported. Especially the inconsistency identification and analyzing for merged requirement- and software architecture models is a creative, communication-intensive, and collaborative task. This paper proposes an approach for an automatic identification and analysis of syntactic and semantic inconsistencies to support distributed modelers who synchronize their parallel work.

Keywords: model driven software engineering, inconsistency management, co-evolution, domain specific modeling

1 Introduction

Currently developed software systems are often large and highly dynamic. Among that, many software systems have to meet several demands as safety-critical reliability, high adaptability, and extensibility at life cycle. These facts have huge impacts on the description techniques. A design or specification document should be intuitively understandable as a matter of communication within an engineering team. But nevertheless, the description should be precisely (formal/computable) formulated to enable a reliable verification. Model Driven Engineering (MDE) approaches are widely-discussed in the scientific community as one answer to the mentioned challenges. MDE is a research approach which focuses on models based on modeling
languages with well-defined abstract syntax and semantics. Further, they provide a
graphical concrete syntax which semantics are intuitive and team-wide
understandable. This is one precondition for a collaborative team-modeling in
creative design phases. But large software systems are often developed in global
software engineering projects [3.]. These kind of projects require a high level of
distributed, collaborative teamwork. Therefore a lot of established configuration
management systems are able to support the realization and integration phases on
source code level. For the management of concurrent evolving text or source code
documents, software configuration management systems (SCM) as CVS [7.] or
collaboration development platforms as [10.,19.] are popular. But similar to source
code, models are subjected to a continuing evolution. In industrial practice a lot of
versions, configurations, or variants are created as account for evolution. The
management of distributed and parallel, evolving models is an ambitious challenge. It
can be divided in three aspects: 1. How is it possible to decompose a complex
distributed model in versionable artifacts? 2. How is it possible to compute a
difference or a merge of parallel evolved models respectively of their versioned
modeling artifacts? 3. How is it possible to preserve the syntactic and semantic
consistency of automatically merged model versions? An answer to the first and
second question is proposed in [4.]. But a pure merge of two parallel evolved models
is an incomplete integration because the consistency of merged models is not proven.
even if two parallel changed model versions are correct regarding syntax and
semantics, the merged result is potentially inconsistent. But a manual inconsistency
resolving in large and complex model repositories is a very time-consuming and
error-prune work. However this process can be supported by computer-aided
inconsistency analysis and visualization.

The problem of consistency-preserving model integration in collaborative software
engineering is well-known and discussed in [13., 20.]. As mentioned in [14.] one way
of concurrent design of models is the persisting of model data as comment-enriched
XML files and a management of these files using SCM systems as CVS or
Subversion. One disadvantage of this kind of systems is the absence of inconsistency
checking of the merge results. Another approach is realized by the teamwork support
in the popular CASE tool MagicDraw [15.]. For consistency preserving, it provides a
pessimistic locking mechanism. But pessimistic locking mechanisms are
unsatisfactory for modeling large models because they restrict concurrent work too
hard [6.]. Some software prototypes which implement merge concepts for
collaborative modeling are published [4., 11.]. However they provide only an
consistency management support for certain modeling languages like UML. To sum
up, there is no so sophisticated computer-aided support for optimistic integration of
domain specific models in collaborative modeling. This paper illustrates a meta-
modeling approach (language design) which provides an automatic inconsistency
detection and analyzing as one phase of the integration of modeling branches.

2 Co-Evolution example scenario

In this section an example scenario for a collaborative and domain specific modeling
process is explained. It enables a vivid depiction for research which will be described in section 4 and 5. The scenario introduces a domain specific model (DSL) and depicts two concurrent modeling branches and an intuitive model merge result.

2.1 A DSL to support the co-evolution of requirements and SW-architecture

The exemplary DSL supports the design of models in early phases of the software engineering process – requirements engineering (RE) and architecture design.

Fig. 1. Excerpt of the ReqArch meta-model

The meta-model-excerpt of the DSL (Fig. 1) describes RE and architectural artifacts and several relations between them. The meta-model is based on the ECore [8]. Especially RE and architecture design are interlaced and require collaborative and pronounced creative modeling activities. Usually a (possibly distributed) team of several software engineers work in parallel on requirements and architecture.

2.2 Concurrent collaborative modeling

A simplified concurrent engineering process is exemplarily illustrated in Fig. 2. The figure shows two software engineers Alice and Bob who adapt independently an origin document version VO and create two revisions VA and VB. With the help of a SCM-
software, VA and VB can be merged. One approach to decompose a model in fine-grained artifacts for the application of such merge mechanism is described in [4]. But the merge result \( \mathcal{VM} \) is potentially inconsistent.

### 2.2.1 Preserving model consistency at integration of modeling branches

To finish an integration of development branches, all inconsistencies have to be identified and resolved. Syntactical inconsistencies in merged models can be identified automatically based on a related formal meta-model. However, semantic inconsistencies cannot be identified in general because the modeling intention of engineers is not automatically analyzable but some kinds of semantic inconsistencies can be automatically detected. A computer-aided analysis of these inconsistencies (e.g. a retrieval for causative model modifications or involved modelers) can support a manual resolution. Furthermore, each inconsistency can be automatically classified related to its violated constraints in the meta-model. All these computable activities can be supported by a computer-aided integration of parallel modeling branches.

![Fig. 3. Example - Merged model \( \mathcal{VM} \)](image)

### 2.2.2 Parallel modifications of requirements and software architecture

Alice and Bob modify dependently an origin model version \( \mathcal{VO} \). The merged model \( \mathcal{VM} \) is shown in Fig. 3. The changes (additions) of Alice and Bob are marked by small green balloons. Alice has extended the origin model with two requirements – #NF-A42 and #F-A40 (additions A0, A1). She has formalized #NF-A42 by an architectural pattern – Layers – (A3) and has bound the introduced roles on all
components (binding is represented by symbols: eye, cog, and DVD – A5-A12). Further, she has specified the semantics of the depend-relation by an example of concatenated representative relations (Usage1 InterfaceRealization – A4). Because of her new requirement #F-A40, Alice has added a new use case PrepareOffer for the system role MarketingSupportSystem (A2). Bob has added a new requirement #F-B17 (B0) and concerning its contents he has added a new use case PrepareOffer (B1). Furthermore, Bob extended the model by the new interface IFBankCounselorClient and two directed relations (usage / interface realization) (B2). The merged model /VM contains several inconsistencies (regarding the meta-model in Fig. 1). The identification and analysis of these inconsistencies is the topic of section 5.

3 Description Logics and their implementations

Description logics (DL) are a family of formal languages for knowledge representation [1]. They resemble a subset of first-order predicate logic. Especially DL provides a more comprehensive expressiveness than propositional logic but is also decidable. A description logic can be divided into two parts: One part (TBox) contains the concept knowledge of the application domain (terminological knowledge) and the second part (ABox) contains a base of valid facts (assertional knowledge). Description logics have well-defined semantics. There are a lot of DL-based systems for computer-aided reasoning: KL-One, LOOM, Fact, Racer, etc. Today, the web ontology language OWL-DL [16] is one of the most popular languages for knowledge representation.

3.1 OWL and conjunctive queries

The proposed approach in this paper uses the OWL2 DL language. It bases on the description logic $SROIQ(D)$. There are several reasons for the design of OWL (2) DL: Maximum of expressiveness possible while retaining computational completeness, decidability, and the availability of practical reasoning algorithms. The proposed approach utilizes some elementary advantages of OWL: OWL has a definite semantics description in opposite to several modeling languages as MOF or UML. There are a lot of efficient reasoners for consistency checking and analysis and there is an implemented draft of a query language which supports conjunctive queries [5].

3.1.1 Reasoning with OWL-ontologies

There are several tools which provide reasoning for OWL ontologies. The most popular tools are Pellet [18], KAON2, and RacerPro. Noteworthy is the fact that all these tools provide interesting checks of consistency and satisfiability:

**Consistency:** An ontology is consistent iff it contains no contradictions. Two examples of inconsistent ontologies (DL-syntax) are: $(\geq 2R \subseteq C , \leq 1R \subseteq C), (\leq 1R \subseteq C , C(a) , C(b) , C(c) , R(a,c) , R(b,c))$
**Concept satisfiability:** A concept $C$ is unsatisfiable iff $C \equiv \bot$. An example for an unsatisfiable concept is: $\neg C \sqsubseteq C$.

These kinds of checks are the foundation for the inconsistency identification and analysis which are discussed in section 5.

### 3.1.2 Analysis of OWL-ontologies by conjunctive queries

Conjunctive queries are one research draft for analyzing OWL-ontologies. Tools like Pellet can process this kind of queries. One important use case for conjunctive queries is a retrieval for individuals with certain characteristics in a knowledge base. For example, the following query requests all components which inherits from itself: $(\text{Component}(x) \land \text{inherit}(x,x))$ If $x$ is a distinguished variable then the answer of the query is $a, b, c$ for the following knowledge base:

$\{ (\top \sqsubseteq \text{Component}, \text{inherit}(\top \sqsubseteq \text{Component}, \text{Component}(a), \text{Component}(b), \text{Component}(c), \text{inherit}(b,a), \text{inherit}(c,b), \text{inherit}(a,c)) \}$

For this example, the query mechanisms enable the identification of all individuals which are involved in a cyclic inheritance relation. Conjunctive queries are the foundation of pre-defined or user-defined inconsistency.

### 3.2 Open vs. closed world assumption

The closed world assumption (CWA) is the dominant paradigm in the area of model driven software engineering. It is the presumption that each statement which is not deducible to be true, is false. But all popular OWL reasoners support only the open world assumption (OWA). The OWA is the presumption that statements are false if and only if they are deducible to be false. There are few ambitions to extend the mentioned reasoners by a CWA support [12.]. The presented approach “emulates” the CWA semantics by explicitly limiting the universe to known individuals and by explicit restrictions for concept and role specifications.

### 4 Design of modeling languages

On the basis of modeling languages, different modelers are able to design an intuitive, generally intelligible, and semantically definite description of specific circumstances. For a consideration of computer-aided inconsistency management, the formal relations between model and its modeling languages play an important role. Furthermore, the presented approach should be applicable for any general purpose and domain specific modeling languages.

#### 4.1 Model syntax definition by Meta Object Facility (MOF)

The Object Management Group provides the standard – Meta Object Facility (MOF) – language for the design of (domain specific or general purpose) modeling languages – especially meta-models. Unfortunately, the MOF specification provides
only informal semantic descriptions. For this reason there are several efforts in the Eclipse Modeling Project (EMF [8.]) to give an implicit definition respectively Java implementation of a formal semantic specification for the essential MOF – EMOF/ECore. This approach based on source code generation from EMF models to Java code based on Java Emitter templates (JET).

4.1.1 Semantic mapping of domain specific meta-models basing on ECore

ECore represents a subset of the MOF meta-meta-model and facilitates the design of domain specific meta-models. The EMF project provides model-to-text transformations from ECore models to Java code. Additional OCL constraints can enrich ECore models to extent their expressiveness. This paper presents an alternative approach. In opposition to Java code respectively OCL, the semantic domain of the proposed mapping is represented by the OWL2 DL expressions. Therefore, it is necessary to implement a transformation which translates ECore models into an OWL2 representation of terminological knowledge (TBox). In the past the EODM project (currently attend by IBM) has implemented such transformation. The realized mapping is sketched in Fig. 4. However, this mapping is nonsatisfying for extensive syntactic checking of models which based on ECore specified meta-models.

One primary disadvantage the EODM-transformation is the incomplete semantic mapping (e.g. attributes like containment and abstract are ignored). Furthermore the transformation proceed on the open world assumption but this fact does not correspond to the generally used closed world semantics of ECore models. For these reasons we implement an alternative and more complete ECore-to-OWL transformation which limits the ontology regarding CWA. The result of the transformation is an OWL-TBox which formalize the semantics of an Ecore-based meta-model. A language designer can manually extent the automatically generated TBox by further OWL-expressions or by a library of conjunctive queries to prove additional constraints of static semantics similar to OCL-expressions. The modeler of the DSL-model (e.g. analyst or architect) needs not be specified any formal expressions to process a syntax check. To illustrate the approach, an exemplary semantic mapping of an excerpt of the introduced example model will be explained in the following. Therefore, Fig. 5 depicts an excerpt of the ReqArch meta model from Fig. 1. It focuses on the containment relation between system roles and use cases.
semantics of the depicted containment relation implies that the maximum one system role can contain an use case. The generated OWL syntax is shown in Fig. 6.

Fig. 6. Containment relation between SystemRole and UseCase – represented in OWL TBox

Besides the semantic mapping of the ECore model ReqArch, the bank modeling which is illustrated in Fig. 3, has to be translated in OWL. Therefore, all instantiated classes, references, and attributes are represented by facts in an ABox. An excerpt of the OWL representation of the example which is illustrated in Fig. 3, is listed in Fig. 7. Within this excerpt, the relations between use cases and roles are described. In [2., 20.] a similar mapping from UML-diagrams to a DL ABox is presented. But in opposite to our TBox-generation approach, these researches presuppose an explicit TBox specification of an (UML) meta-model for consistency checks.

For a syntactical consistency check of the collaboratively developed bank model in the example scenario, the generated OWL expressions of the TBox (Fig. 6) and the ABox (Fig. 7) are joined together in one ontology. Section 5 explains how reasoning and query mechanisms enable this kind of analysis. However, for processing a semantic analysis of the domain specific model, a alternative knowledge base has to be generated. This translation is sketched in the next subsection.

4.1.2 Semantic mapping of domain specific models

In opposite to the semantic mapping of ECore models, the semantics of a domain specific model cannot be

Fig. 6. Containment relation between SystemRole and UseCase – represented in OWL TBox

Besides the semantic mapping of the ECore model ReqArch, the bank modeling which is illustrated in Fig. 3, has to be translated in OWL. Therefore, all instantiated classes, references, and attributes are represented by facts in an ABox. An excerpt of the OWL representation of the example which is illustrated in Fig. 3, is listed in Fig. 7. Within this excerpt, the relations between use cases and roles are described. In [2., 20.] a similar mapping from UML-diagrams to a DL ABox is presented. But in opposite to our TBox-generation approach, these researches presuppose an explicit TBox specification of an (UML) meta-model for consistency checks.

For a syntactical consistency check of the collaboratively developed bank model in the example scenario, the generated OWL expressions of the TBox (Fig. 6) and the ABox (Fig. 7) are joined together in one ontology. Section 5 explains how reasoning and query mechanisms enable this kind of analysis. However, for processing a semantic analysis of the domain specific model, a alternative knowledge base has to be generated. This translation is sketched in the next subsection.

4.1.2 Semantic mapping of domain specific models

In opposite to the semantic mapping of ECore models, the semantics of a domain specific model cannot be
specified independently of the DSL. It is rather one part of the DSL definition. In this subsection, the semantic mapping for the introduced example (Fig. 3) is exemplarily illustrated. Goal of the mapping is the semantics definition based on an OWL TBox representation to analyze the model semantics regarding inconsistencies.

The model excerpt of the bank example focuses on the changes which are marked by A3, A4, and B3 in Fig. 3. The OWL representation of Bob’s new interface relation (B3) is depicted in Fig. 9. The topic of the following section is the identification and analysis of these inconsistencies.

5 Inconsistency management based on an OWL representation of models

The generated OWL representation of models enables an inconsistency analysis by reasoning tools. Therefore the mentioned reasoning services (Section 3.1.1) can be applied. The illustrated model excerpt of Fig. 3 contains two inconsistencies, one syntactic and one semantic inconsistency which are discussed in the following subsections.

5.1 Syntactic analysis

The containment relations between the use case PrepareOffer and the system roles CustomerManagement and MarketingSupport violate the meta-model constraint which is illustrated in Fig. 5. This syntactic inconsistency can be identified by an OWL reasoner (cf. Subsection 3.1.1). The syntactic analysis for the presented and some further examples was evaluated by the Pellet reasoner. The following listing shows the output of Pellet analyzing the mentioned knowledge base.

```
WARNING: Inconsistent ontology.
Reason: Individual http://bank/PrepareOffer has more than 1 values for property
http://bank.de#isContainedBy violating the cardinality restriction
```

The reasoner detects and analyses one inconsistency. It identifies that the use case PrepareOffer is contained by more than one individual. After the identification and
initial analysis, an inconsistency can be analyzed by conjunctive queries on the generated ontology. In the presented example the use case PrepareOffer is contained by “more than 1” roles. Alice and Bob ask themselves: Which systems contain the use case PrepareOffer? Sure, in a small model the question is easy to answer but not in large model repositories. For this case, they can use (conjunctive) queries to analyze the detected inconsistency. The following listing (SPARQL [17.] syntax) shows the mentioned question in head and the answer of the Pellet engine afterwards.

```sparql
SELECT DISTINCT ?X WHERE
  { <http://bank/PrepareOffer> bank:containedBy ?X . }
```

| X                                    |
|========================================|
| <http://bank/CustomerManagement>     |
| <http://bank/MarketingSupportSystem> |

Accordingly, PrepareOffer is contained by CustomerManagement as well by MarketingSupportSystem. After the analysis Alice and Bob have to resolve the inconsistency in a collaborative way. Besides a syntactic analysis, a semantic analysis can be operated.

### 5.2 Semantic analysis

The basis of a semantic analysis is a OWL TBox representation of the model (cf. Fig. 1, Fig. 9). To illustrate a semantic analysis of the merged bank model, the changes A3, A4, and B3 are focused in the following. Alice has defined a new architectural pattern (A3) and has specified the semantic of the depend-relation by two instances of the relations Usage and InterfaceRealization. Therefore, Alice’s specification implies that the depend relation is interpreted as a concatenated relation of Usage and InterfaceRealization. But the usage relation from CustomerFile to BankCounselorClient which was added by Bob is in conflict with the approved dependencies. This is a semantic inconsistency and it can be detect by a class satisfiability check of an OWL reasoner as listed in the following output:

```
Reasoner Report
Warning (Unsatisfiable class): http://bank.de#IFBankCounselorClient
```

Because of the unsatisfiable depend relation, an individual (instance) of type IFBankCounselorClient cannot exist. In the exact same manner of syntactic analysis, the identified inconsistency can be analyzed more comprehensively with conjunctive queries.

### 6 Conclusions and further work

The paper proposes an approach for a syntactically and semantically formal representation of models in model driven software engineering based on OWL2. This representation enables the presented computer-aided inconsistency identification and analysis. Therefore, popular OWL2 reasoners as Pellet and KAON2 provide consistency and concept satisfiability checks which can be used for inconsistency
manangement. Furthermore, the paper presents an extended analysis of inconsistencies by conjunctive queries on the OWL representation of a merged model. The further work is structured into two areas: The generation of an OWL representation from a (meta-) model will be refined to cover the whole ECore meta-meta-model. Furthermore, other representation approaches especially for CWA modeling [12.] should be evaluated for a semantically precise representation of models. On the other hand, the analyzing mechanisms providing by ontology debugging approaches should be evaluated regarding aptitude for inconsistency management in model driven software engineering.

References

16. Web Ontology Language (OWL), http://www.w3.org/2004/OWL/
Model Refactoring in Eclipse by LTK, EWL, and EMF Refactor: A Case Study

Thorsten Arendt¹, Florian Mantz¹, Lars Schneider², Gabriele Taentzer¹

¹ Philipps-Universität Marburg, Germany, FB 12 - Mathematics and Computer Science, Software-Engineering
{arendt,mantz,taentzer}@mathematik.uni-marburg.de
² Capgemini sd&m, Offenbach, Germany derlarsschneider@googlemail.com

Abstract. Since model-driven development (MDD) has evolved to a promising trend in software development, models become the primary artifacts of the software development process. To ensure high model quality, using appropriate quality assurance techniques like model refactoring is an essential task. So far, tool support for model refactoring is limited, particularly for models using the Eclipse Modeling Framework (EMF). In this paper we present the results of a small case study that examines three solutions for a sample EMF model refactoring, namely the Language Toolkit (LTK), the Epsilon Wizard Language (EWL) and EMF Refactor, a new Eclipse plug-in for EMF model refactoring.

1 Introduction

Model-driven development (MDD) has become a promising trend in software development. Here, models are in the focus of work and represent the primary artifacts in the software development process. Considering code generation, software quality depends directly on the quality of input models. Furthermore, the Unified Modeling Language (UML) evolved to a quasi-standard to be used to develop high quality models.

To obtain high quality models, the existing model quality has to be determined regarding selective quality aspects of interest. During model evolution, an ongoing revision of the model quality is required. An obvious approach for quality assurance of UML2 models is to lift software assurance techniques to the level of models where possible. Here, well-known techniques like software metrics, code smells, and code refactorings [10] have been taken into account.

A variety of tools for quality assurance of code exist, in particular for the refactoring of Java code. For model refactoring, however, tool support is limited so far. Since the Eclipse Modeling Framework (EMF) [1] has become a key reference in the field of MDD, it is obvious to adapt tools supporting quality assurance techniques for EMF models.

In this paper, we present three approaches for specifying and applying EMF model refactorings. We specify a sample UML2 model refactoring by means of the Language Toolkit (LTK) [6] and the Epsilon Wizard Language (EWL) [4], two existing solutions to handle refactorings in Eclipse. Furthermore, the case
study investigates a new EMF model refactoring tool, called EMF Refactor [2] which relies on EMF Tiger [3], a model transformation tool based on graph transformation concepts. The approaches are analyzed with respect to seven defined evaluation criteria. As a result, the different ways of specifying and executing a selected model refactoring are discussed and the benefits as well as drawbacks of each approach are pointed out. EMF Tiger is not in the focus of this study. In [15], however, EMF Tiger and a number of other graph transformation-based tools were compared with each other using a compact practical model transformation case study.

This paper is organized as follows: In Section 2, the evaluation criteria for the case study and the selected model refactoring \textit{Change Attribute to Association End} are presented. Section 3 briefly explains how the selected refactoring is specified using the investigated approaches. Their benefits and drawbacks are discussed in Section 4. In Section 5, we summarize our contributions and discuss future work.

2 Case Study Description and Evaluation Criteria

This section introduces the sample UML2 model refactoring \textit{Change Attribute to Association End} and specifies the evaluation criteria for the case study.

2.1 Sample Refactoring

Since UML2 class diagrams are very closely related to source code, many existing code refactorings can be directly adopted to UML2 class diagrams. However, there are few model refactorings which are specific to the model level and therefore cannot be adopted from code refactorings. The sample refactoring used in this case study is one of the latter category which changes an attribute to an association end.

![Fig. 1. Sample Class Diagram before Refactoring (excerpt)](image)

Fig. 1 shows an excerpt of a class diagram. At a first glance, one might suppose that class \textit{Address} is isolated from all other model elements. But if we take a closer look to the model, we identify attribute \textit{address} in class \textit{Customer} which is of type \textit{Address}.

For a better understanding of class structures, it would be worthwhile to represent this relationship more explicitly. This can be achieved by applying model refactoring \textit{Change Attribute to Association End}. After refactoring application, attribute \textit{address} of class \textit{Customer} will be depicted as an association end (see specification of UML2.1 [5]).
2.2 Evaluation Criteria

Each approach is investigated considering the following evaluation criteria. For each criterion questions are defined which are evaluated during the specification and execution of sample refactoring solutions.

Refactoring Specification

- **Complexity** - How complex is the effort to specify the refactoring? Are there ways to reduce this effort? Here, LoC and the number of specified rules have to be considered.
- **Correctness** - Is it possible to specify a refactoring which application results in an inconsistent model? Are there any precautions to avoid it?
- **Testability** - Which effort is needed to test the specified refactoring in detail? Are there ways to automate these tests?
- **Modularity** - Can the specified refactoring be combined with other refactorings? (This is an important aspect when defining more complex refactorings by reusing existing ones.)

Refactoring Application

- **Interaction** - How convenient is the application of a refactoring? Are there any facilities to simplify user inputs? Here, differences considering UI features have to be evaluated from a (subjective) user’s point of view.
- **Features** - Does the refactoring provide a preview in order to be able to cancel or commit the refactoring? Does it provide undo and redo functionality?
- **Malfunction** - What happens if the appropriate refactoring cannot be executed in a given situation? Are there reasonable error messages?

3 Case Study Execution

The sample model refactoring was implemented using LTK, EWL and EMF Refactor. Due to space limitations, the implementations are presented in a very compact form. The entire specifications (source code, rules, etc.) can be found on the EMF Refactor web site [2].

3.1 Refactoring Implementation using LTK

The Language Toolkit (LTK) [6] is a language neutral API to specify and execute refactorings in an Eclipse-based IDE. So it is possible to handle EMF model refactorings by LTK. The API can be found in the `org.eclipse.ltk.core.refactoring` and `org.eclipse.ltk.ui.refactoring` plug-ins. The API classes of LTK incorporate an exact, predefined procedure for refactorings in Eclipse.
For specifying the sample model refactoring, 7 classes have to be implemented. During implementation it became obvious that only 4 classes are refactoring specific (RefactoringInfo, RefactoringInputWizardPage, RefactoringAction, and RefactoringProcessor). Classes EMFChange and Refactoring are generic for EMF model refactorings. Class RefactoringWizard is refactoring specific only, since it initializes RefactoringInputWizardPage. The refactoring specific classes are:

- **RefactoringInfo** - This class manages all required informations like the selected `Property` object, the name of the new association and the name of the association's `ownedEnd` property.
- **RefactoringInputWizardPage** - This class is responsible for displaying and handling the required user input (name of the new association and name of the association's `ownedEnd` property).

```java
RefactoringStatus result = new RefactoringStatus();

if (property.get(property) != null) {
    if (property.get(Property).isInstanceOfClass) {
        if (this.property.get(Property).getAssociation() != null) {
            result.addFatalError("The selected Property is already an association end!");
        } else {
            result.addFatalError("The name of the selected Property is not a Class!");
        }
    } else {
        result.addFatalError("The selected Property does not have a type!");
    }
    return result;
}
```

Fig. 2. LTK: method body `RefactoringProcessor.checkInitialConditions()`

- **RefactoringAction** - This class is responsible for refactoring initiation. It sets the selected `Property` and initializes instances of RefactoringWizard, RefactoringProcessor, Refactoring, and RefactoringInfo. The refactoring is initiated by invoking method `RefactoringWizardOpenOperation`::run(). The extension point `org.eclipse.ui.popupMenus` is served by this class.

```java
Map.Entry<FeatureChange, Entry<FeatureChange>> entryAsName = createEObjectToChangesMapEntry(as);

FeatureChange fCasName = createFeatureChange();

fCasName.setFeatureName("name");

fCasName.setValue(this.property.getAssociation().asName());

entryAsName.setValue(add(fCasName));

changeDescription.getEObjectChanges().add(entryAsName);
```

Fig. 3. LTK: method `createChange()` (excerpt)

- **RefactoringProcessor** - This is the main class for executing the sample refactoring. Method `checkInitialConditions()` checks whether the type of the selected `Property` is an instance of `Class` and whether it is not already part of an `Association` (see Fig. 2). The most important method of class `RefactoringProcessor` is `createChange()`. This method creates an instance of `EMFChange` by generating a `ChangeDescription` that describes

---

3 `org.eclipse.ltk.ui.refactoring.RefactoringWizardOpenOperation`

4 `org.eclipse.emf.ecore.change.ChangeDescription`
all required model changes and is also used for undo and redo functionality. Fig. 3 shows an excerpt of method `createChange()`. Here, feature `name` of the newly created `Association` is set to the appropriate String managed by the `RefactoringInfo` object.

3.2 Refactoring Implementation using EWL

The Epsilon Wizard Language (EWL) is an integral part of Epsilon [13], a platform for building consistent and interoperable task-specific languages for model management tasks. For this purpose, Epsilon consolidates common facilities in a base language, the Epsilon Object Language (EOL) [11], that new task-specific languages can reuse.

EWL is a tool-supported language for specifying and executing automated model refactorings which the authors of EWL call update transformations in the small [12]. These model refactorings are applied on model elements that have been explicitly selected by the user. Epsilon provides an Eclipse-based interpreter that can execute programs written in EWL.

In EWL, the sample refactoring has been implemented as follows: First, the type of the selected model element has to be checked to be a `Property` of type `Class`. Furthermore, this property does not already have to be part of an `Association`. These preconditions are checked in the `guard` section of the EWL program. Variable `self` refers to the model object which is used to invoke the refactoring and is a `Property` in this example. If the guard conditions fail, the refactoring will not be performed. Fig. 4 shows the `guard` section of the EWL solution.

The next step is to specify the label that will be provided to the user in the context menu of the selected model element. This is done in the `title` section of the EWL program.

The final and most important part of the EWL solution is the `do` section that specifies the effects of the refactoring when applied to a compatible selection of model elements (see Fig. 5). Regarding the sample refactoring we have to organize the required user input first, in particular the name of the new association and the name of the association’s `ownedEnd` property.

After obtaining the user input all necessary new objects are created and the appropriate features are set. These are in particular:

```
var upperVal : new LiteralInteger;
upperVal.value = 1;
var lowerVal : new LiteralInteger;
lowerVal.value = 1;
var ownedEndP : new Property;
ownedEndP.name = scProperty;
ownedEndP.type = self.class;
ownedEndP.upperValue = upperVal;
ownedEndP.lowerValue = lowerVal;
var asso = new Association;
asso.name = associationName;
asso.ownedEnd.add(ownedEndP);
self.class.package.packagedElement.add(asso);
```
– A new Property with features name, type, upperValue, and lowerValue (that are set to 1 each).
– A new Association with features name, ownedEnd, and memberEnd.

Again, global variable self is used to get the appropriate features of the selected Property. Finally, the new association has to be added to the including package.

3.3 Refactoring Implementation using EMF Refactor

A new approach to specify and execute EMF model refactorings is EMF Refactor [2]. The development of new refactorings in EMF Refactor is based on EMF Tiger [3] [8], an Eclipse plug-in that performs in-place EMF model transformations [7] [14]. The model transformation concepts of EMF Tiger are based on algebraic graph transformation concepts. It provides a graphical editor for the design of transformation rules and a Java code generator which has been extended by EMF Refactor.

Model refactorings are designed by ordered sets of rules. Each rule describes an if-then statement on model changes. If the pattern specified in the left-hand side (LHS) exists, it is transformed into another pattern defined in the right-hand side (RHS). Here, several input parameters can be used to specify the LHS pattern in more detail. Additionally, several negative application conditions (NACs) can be specified which represent patterns that prevent the rule from being applied. Mappings between objects in LHS and RHS and/or between objects in LHS and NACs are used to express preservation, deletion, and creation of objects. A LHS object being mapped to a RHS object is preserved, while an object without mapping to a RHS object is deleted from the model including all its possible children. A RHS object without an original LHS object is newly created and attached to the model.

![Fig. 6. EMF Refactor: LHS](image)

To specify the sample refactoring in EMF Refactor we have to define one rule. The LHS of this rule is shown in Fig. 6. This pattern represents the abstract syntax which has to be found when starting the refactoring from within the context menu of a Property named propName whose type is a Class. To ensure that the selected Property is not already part of an Association an appropriate NAC is defined, that is similar to the LHS but with an additional Association instance that references the selected Property as memberEnd (not shown here).

Fig. 7 shows the RHS of the sample refactoring rule. It contains a new Association object with a new opposite association end (Property). This end
is equipped with multiplicity 1 as lower and upper bound. The newly created objects are named by additional input variables `associationName` and `srcProperty`.

The rule specification ensures that the specified transformation rule is consistent. This means that the application of an appropriate EMF model transformation always leads to EMF models consistent with typing and containment constraints. To do so, you have to check whether the rules perform restricted changes of containments only. Consistent EMF model transformations behave like algebraic graph transformations. Hence, the rich theory of algebraic graph transformation can be applied to show functional behavior and correctness [9]. The sample refactoring rule is consistent, since all new object nodes (Association, Property, and two LiteralIntegers) are connected immediately to their according container (see Fig. 7).

After rule definition the corresponding refactoring code is generated, including a wizard for parameter specification. Here, default values for the parameters `associationName` and `srcProperty` are set.

4 Case Study Results and Evaluation

This section presents the results of the case study. First, all three solutions are compared along the criteria introduced in Section 2 and finally they are interpreted.

Complexity - All approaches require a comprehensive understanding of the UML2 meta model [5]. In LTK, 7 Java classes including 711 LoC were implemented. 416 LoC can be generated and 195 are refactoring specific, in particular methods `checkInitialConditions()` and `createChange()` of class `RefactoringProcessor`. Here, the most challenging task is to exactly implement the corresponding `ChangeDescription` object because of its general and complex API. In EWL, one single file with 47 LoC was implemented. Automatically generating generic parts would not lead to a significant reduction. Finally, in EMF Refactor the whole refactoring code was generated from one rule only, containing 32 ob-
jects (EClass and EReferences). Individual parameter settings for code generation are supported by a convenient wizard.

**Correctness** - In LTK, an incorrectly specified ChangeDescription object would lead to an inconsistent model after executing the refactoring. There are no known precautions available to avoid this. Since all model changes in EWL are directly implemented, there is also no special support to specify refactorings which yield consistent models only. EMF Refactor instead uses EMF Tiger that provides consistency checks regarding containment and multiplicity issues, incorporated in its visual editor. Hence, it is almost impossible to specify transformations, especially refactorings leading to inconsistent models.

**Testability** - A specified refactoring has to be tested by applying it to various models that represent possible situations. Since every refactoring in LTK is a single Eclipse plug-in, it is very time-consuming to start a new Eclipse instance after each code change. This task could be significantly facilitated by generating test code or using PDEUnit which is a test framework for Eclipse plug-ins. Since EWL is an interpreted language, testing is not that time-consuming, but a straightforward task. However, there is no known way to automate these tests. For EMF Refactor the same comments as for LTK hold. Here, a first approach for generating tests using JUnit is available.

**Modularity** - Since all model changes in LTK are directly implemented in Java, it seems to be possible to combine several existing refactorings to more complex ones by passing required parameters and adapting conditions, using class ChangeDescription. Here, it is necessary to develop an advanced approach to support this features. In EWL, there is no known way to combine refactorings so far, except for copying and adapting code of existing ones. For EMF Refactor the same comments as for LTK hold. A first approach to combine so-called basic refactorings to more complex ones is under development.

**Interaction** - All approaches provide the selection of refactorings by the context menu of a Property element in the standard EMF instance editor. EWL additionally supports UML2Tools which can be supported by the others as well if a further extension point is served. The refactoring wizard page of LTK provides one input line for each required parameter. Each parameter has a specified default value. Using EWL, the context menu has an entry named specifically according to the name of the selected Property. All parameters are entered in separate dialogs including specified default values. For EMF Refactor the same comments as for LTK hold.

**Features** - After parameter editing in LTK, the wizard provides an optional preview of the model changes made by the refactoring. The preview is provided by EMF Compare. Undo/Redo functionality is also supported. In EWL, there is no preview available, but Undo/Redo functionality is supported. After parameter editing in EMF Refactor the wizard always shows a preview of potential model changes when executing the refactoring. Again, this is provided by EMF Compare. Undo/Redo functionality is not supported.

**Malfunction** - If a certain precondition in LTK fails, a message box including a reasonable error message is shown as specified in method checkInitial
Conditions() of class RefactoringProcessor. EWL provides the refactoring only, if all preconditions specified in the guard section hold. After parameter input in EMF Refactor, the user is informed when the refactoring can not be executed because of violated conditions. This is merely done by the generic message The refactoring changed nothing at all. For the considered example, each solution requires non-empty parameters, i.e. names for the new model elements Association and Property.

<table>
<thead>
<tr>
<th>Goal</th>
<th>LTK</th>
<th>EWL</th>
<th>EMF Refactor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>o</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Correctness</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Testability</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>Modularity</td>
<td>o</td>
<td>-</td>
<td>o</td>
</tr>
<tr>
<td>Interaction</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Features</td>
<td>+</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>Malfunction</td>
<td>+</td>
<td>+</td>
<td>o</td>
</tr>
</tbody>
</table>

Table 1. Results of the Case Study

Table 1 summarizes the results of the case study. Each approach has been evaluated and marked as follows:

- The approach meets the evaluation criterion: +
- The approach does not meet the evaluation criterion but is still moderate: o
- The approach does not meet the evaluation criterion at all: -

Each approach has its individual strengths and weaknesses. LTK provides permanent positive results when executing the model refactoring. This is not astonishing, because LTK was developed to unify refactoring processes in Eclipse. However, EMF Refactor seems to be more suitable for specifying EMF model refactorings. This is because of its graphical nature of defining model transformations and its underlying graph transformation concepts. Last but not least, EWL shows advantages in both categories, refactoring specification and application. However, in both categories there is another approach each that seems to be more suitable than EWL.

5 Summary and Future Work

In this paper, we present the results of a small case study that examines three options for EMF model refactoring, namely the Language Toolkit (LTK), the Epsilon Wizard Language (EWL) and EMF Refactor, a new approach in the field of EMF model refactoring. The study demonstrates that each approach has its individual strengths and weaknesses. LTK is the leading approach during model refactoring application, whereas EMF Refactor seems to be the most promising one in specifying EMF model refactorings.

It is recommended to consider the results of this case study in future work on model refactoring tools, and to extend it by specifying further refactorings. Several extensions concerning Testability and Modularity are under development,
especially for \textit{EMF Refactor}. Moreover, undo/redo functionality and the allocation of reasonable error messages have to be considered for \textit{EMF Refactor}. As a conclusion of the presented case study, it looks worthwhile to check whether \textit{LTK} can be combined with an EMF-related refactoring tool in a way that merges the benefits of both approaches. At least, a combination of \textit{LTK} with \textit{EMF Refactor} seems to be a promising way to go.

References

Implementing An Inconsistency Management Framework for Model Transformation Tools

Olaf Muliawan, Dirk Janssens
Universiteit Antwerpen
Antwerpen, Belgium
{olaf.muliawan, dirk.janssens}@ua.ac.be

Abstract. Although the correctness of model transformations is a legitimate concern, the correctness of the model transformation tools themselves is rarely discussed. To see if such tools are correct developers use ad-hoc testing and debugging to discover any inconsistencies. However, a comprehensive representation of inconsistencies for model transformation tools is lacking. Therefore, we adopt earlier work from the MoVES consortium: a proposed general framework for inconsistency management, specifically applied on model transformation tools. In this paper we explain the features of this framework and its instantiation in the case of model transformation tools. We also explore the benefits that an explicit model for inconsistencies offers for the development of correct tools. We illustrate our findings with MoTMoT, the Model To Model Transformation Tool developed at the University of Antwerp.

1 Introduction

The development of model transformation tools focuses on correctness at two levels: first, there is the correctness of the transformations the tool produces and second, there is the correctness of the tool itself. A number of different transformation tools already ensure that transformations have an input and output corresponding to their meta-model, e.g. through the use of OCL rules as preconditions to validate input models. However, how do developers know that the transformation tool performs as expected? Testing and debugging are techniques to check if the transformation tool produces the expected results, though this only provides a partial image of inconsistencies. The cause and the exact location of the inconsistencies are not immediately shown.

Within the MoVES\textsuperscript{1} consortium we have cooperated with several partners to come up with a framework to model the inconsistency management of software [1]. Our work is based upon earlier findings on inconsistency management, such as the viewpoints framework by Finkelstein et al. [2] and its extension by Nentwich et al. [3]. In this paper we will implement the inconsistency management

\textsuperscript{1} This work is published for the consortium for Fundamental Issues in Modelling, Verification and Evolution of Software (P6/39), part of the IAP-Phase VI Interuniversity Attraction Poles Programme funded by the Belgian State, Belgian Science Policy.
framework at the level of the transformation tool. The framework allows us to analyse our tool more thoroughly and uncover errors more easily.

Our contribution focuses on the management of inconsistencies in the field of model transformations. If we want to resolve inconsistencies, an explicit description, rather than an informal assessment, is more precise and allows for better collaboration with other colleagues. This is especially the case when analysis involves external tools responsible for checking correctness, model storage, visual representation of the model, ...

The paper is structured as follows: in the next Section we discuss the transformation tool MoTMoT and shed a light on its possible inconsistencies. In Section 3 we explain the details of the consistency framework and show its implementation for MoTMoT. Finally, we provide a conclusion in Section 4 with an outlook on the use of the framework.

2 A visual graph transformation tool

The field of model transformations has matured over the years and a number of transformation languages have been developed. Some tools, such as ATL, pursued a declarative, rule-based approach. Other tools based themselves on graph transformation techniques and presented model transformations in a visual manner. One of these tools is called MoTMoT and was developed at the University of Antwerp. More specifically, it employs UML diagrams, such as class and activity diagrams, to represent model transformations [4]. To achieve consistency, we need to screen these diagrams on their syntactic consistency and this requires a pre-processing step before compilation where diagrams are validated.

The syntax of these diagrams is defined in the language of Story Driven Modeling (SDM). This is a model transformation language supporting an imperative control flow for graph transformations [5]. In MoTMoT, activity diagrams represent the control flow while class diagrams represent the graph patterns of model elements. A UML profile for Story Driven Modeling defines the concrete syntax using stereotypes and tagged values on class diagram and activity diagram elements.

Evaluating consistency can be done on two levels: on the one hand, the specification level where the transformation is modeled using a mix of UML class and activity diagrams. On the other hand we have the execution level where an input model is processed and transformed resulting in an output model. At the specification level preconditions are verified to see if the transformation is modeled according to the SDM language, at the execution level the transformation is executed and sample input models can be checked on their properties.

The principle goals of MoTMoT are support of existing OMG standards and reuse of existing frameworks. This is why we use AndroMDA: this is a code generation tool which transforms the visual SDM model in UML into an executable transformation as Java source code [6]. The MoTMoT transformation flow is presented in Figure 1. The first phase consists of generating the transformation code: a visual MoTMoT model of the transformation is specified first.
AndroMDA analyzes the transformation and retrieves the necessary information to generate transformation code in Java. In a second phase, the transformation is executed: first the OCL evaluation engine performs an analysis on the input model and then transforms it according to the specification in the transformation model.

AndroMDA employs the facade pattern to wrap meta-model elements. The facades (called meta-facades) provide a link between the wrapped meta-model element and AndroMDA, the possibility of implementing custom operations and, important in our case, the facility to impose consistency conditions as constraints in OCL (Object Constraint Language). For the actual code-generation AndroMDA uses a template engine. This engine will access the meta-facades and employ the helper methods to fill in the textual template.

We see an example of these constraints in the two meta-facades in AndroMDA, shown in Figure 2. We have OperationFacade, which encapsulates the UML meta-element Operation, and the inherited MotMotOperationFacade, which specifies extra helper methods and imposes additional restrictions on instances of Operation. In this case, the return type and parameters of the operation require valid names. This consistency condition is specified as an OCL constraint in AndroMDA.

As seen in Figure 2, these consistency conditions are “hidden” in the transformation specification. A more explicit description of these rules, with regards to all involved stakeholders, is desirable.
3 Management of inconsistencies

As part of our work in MoVES, we have identified three key aspects in the field of inconsistency management [1]. The first of these is *consistency specification*: programming or modeling languages require explicit support to define consistency conditions. At run-time *inconsistency detection* tools determine if all rules are validated successfully or, if some are violated, it indicates inconsistencies. The third area focuses on *inconsistency handling*: in the event an inconsistency is detected, should these inconsistencies be automatically resolved or flagged for manual resolution? Another resolution strategy could even ignore some inconsistencies: e.g. if consistency conditions for a given model merely indicate which structures could be improved, violation of these rules is not fatal but a sign that the given model can be modified into a better structure. Egyed published work on the instant evaluation of UML diagrams for their consistency [7], however his work did not broaden the scope on the underlying causes of inconsistencies like this framework does.

The framework is presented in Figure 3, where the three key aspects are visible. The main goal of the instantiations of this framework is an explicit description of the languages involved and their constraints and models. Note that these areas overlap, especially on the (consistency) conditions themselves:

- **Consistency specification**: The *consistency conditions* are defined in one or more *condition languages* (e.g. Well-Formedness Rules in OCL and assertions in Java). Each of these conditions is defined on a given *model*. The model itself is defined in one or more *modelling languages* (e.g. models defined as Petri nets and UML activity diagrams). However, while conditions are checked on models as a whole, the conditions are only expressed in terms of model elements. For this reason it is important to separate the level of models and their model elements.
– **Inconsistency detection**: The evaluation of the conditions occurs over a set of given models. During detection we return the individual model elements involved in the inconsistencies. In addition, these inconsistencies are correlated to the violated conditions.

– **Inconsistency handling**: When inconsistencies are detected, the tool or the developer determines how to resolve the inconsistencies.

![Diagram of general framework for the management of inconsistencies](image)

To illustrate the framework for the management of inconsistencies, we have chosen to provide the following instantiation of the framework: we show how MoTMoT interacts with AndroMDA, a model-to-text transformation tool, to generate a transformation file in Java. Inconsistencies in the conversion from model to text are handled using an OCL interpreter within AndroMDA. This instantiation shows the three key aspects of the framework, focusing on the consistency conditions, the inconsistency detection and the inconsistency handling.

### 3.1 Instantiation: Managing syntactic inconsistencies in SDM diagrams

The tool MoTMoT can only parse correct SDM diagrams. For this reason we use OCL constraints to express a strict consistency between the MoTMoT transformations and the SDM meta-model. More specifically, in MoTMoT we cannot tolerate any inconsistency and consistency resolution is focused on fixing the elements involved in the constraint to achieve a consistency.

In this illustration we focus on just a selected number of constraints, however in MoTMoT they are all constructed in a similar manner. OCL constraints are defined on UML class diagrams and activity diagrams, so the following constraints will only involve those language elements.

– Each action state (representing an activity) in the UML activity diagram should be linked to a UML class diagram. The class diagram contains the
description of the graph transformation. This is valid for all states, except when the state is a code state (containing Java code instead) or a link state (referring to another control flow):

\[
\text{context MotMotActionStateFacade inv: isTransPrimitiveState() implies hasTransPrimitivePackage() = true}
\]

So for each action state in the activity diagram and if that state is supposed to contain a transformation diagram, we check if that state indeed has a link to a class diagram.

- All classes within a class diagram, which are part of a graph transformation, should have a type. The classes represent typed nodes, the existence of untyped nodes is prohibited. This is expressed by the constraint:

\[
\text{context MotMotClassFacade inv: isPartOfTransformation() implies getTypeName() <> ""}
\]

So, for each class which is part of a transformation pattern we check whether they have a type attached to it.

- A more complex constraint involves the matching algorithm of MoTMoT. To successfully match all nodes, every node requires a connection to a \textit{bound} node (a node which has already been identified in an earlier matching phase). If such a connection is missing, MoTMoT cannot generate a transformation. Due to limited space, the constraint cannot be shown, but it includes the following checks: see if all ends are navigable (i.e. all elements can be found), ensure that \textit{bound} nodes were indeed identified before and compare if connections in the transformation have a counterpart in their meta-model (i.e. the connections exist). The constraint, including the helper functions, consists of 40 lines. This illustrates that the constraints in MoTMoT are not all trivial.

(a) UML Activity Diagram

(b) UML Class Diagram

Fig. 4: Illustration of incorrect diagrams in MoTMoT
Observe Figure 4a where we show an incorrect UML activity diagram. The state named `Create List Impl` is missing a tagged value to denote it is connected to a class diagram. All other states in the diagram do have a connection with a tagged value `motmot.transprimitivepackage`, a link to the UML package containing the matching class diagram. This is only one example which indicates that the consistency of this diagram is required to generate a correct transformation. An incorrect UML class diagram is shown in Figure 4b: a connection is missing between `activityA` and the other elements and the latter cannot be found with the matching algorithm.

**Framework instantiation** The instantiation of the framework can be seen on Figure 5. The consistency specification is defined as:

- The set of models (UML class and activity diagrams) considered for detecting inconsistencies, called *SDM diagrams*.
- The set of modelling languages in which these models are expressed. In this case, this is both UML and Java. SDM diagrams consist of UML class and activity diagrams. The consistency conditions are expressed in terms of UML classes, tagged values and other relevant UML elements.
- The set of consistency conditions defined over those models as OCL constraints.
- The set of condition languages contains both OCL and Java. We use *AndroMDA* to evaluate OCL constraints. The evaluation of constraints in Java involves boolean expressions. As the OCL evaluation engine in AndroMDA is limited, we include support for Java expressions. Due to the difficulty in determining if these latter expressions are correct, the use of Java is discouraged (and should phase out with the use of a more powerful OCL evaluator).
- The set of OCL constraints are defined on UML class and activity diagram classes. The model contains instances of involved UML language elements like `Class`, `Association`, `ActionState`, `TaggedValue`, etc.

**Consistency conditions** The inconsistencies we want to avoid are violations of the SDM syntax. For example, consider a class diagram representing a graph pattern for MoTMoT. Each class represents a node in the graph pattern. Each class requires a model element type tag which indicates the represented model element. The conditions are specified in the Object Constraint Language (OCL) and deal specifically with the syntactic structure of model transformation specifications. These conditions are checked on UML activity and class diagrams. Moreover, these diagrams have to conform to the UML language (this is usually enforced through the UML diagram tool) and the OCL conditions for MoTMoT transformation models. Structural inconsistencies are flagged and shown to the user. Due to the nature of the transformation tool, user input is required to fix inconsistencies and automatic consistency resolution is not possible.

**Inconsistency detection** There are basically two ways to enforce consistency on models representing model transformations. The UML profile for SDM restricts
Inconsistency specification

Inconsistency detection

Inconsistency handling

SDM diagrams defined on A class, a state, a tagged value, ...

UML class and activity diagram meta-classes

OCL constraints expressed in terms of

UML, Java (AndroMDA)

OCL, Java (AndroMDA)

composed of

expressed in

instance of

expressed in

expressed in terms of

expressed in terms of

Consistency Specification

Inconsistency Detection

Inconsistency Handling

Fig. 5: Framework instantiation for Model Transformation specifications

users in their use of UML model elements: only a subset of model elements can be used, therefore limiting the language. OCL constraints impose more syntactic conditions on the models. For example, every state in the control flow of the transformation should contain a graph rewriting. An OCL constraint could impose this requirement.

These constraints have to be verified and AndroMDA has an OCL validation tool built in. Before translation, a validation phase is run and if the model cannot be validated a set of violated constraints is shown, along with a comment explaining the error and possibly offering a suggestion to fix the violated constraint.

At the moment we do not check whether all loops terminate: only finite sets of model elements are successfully matched. However, future work will focus on issues of termination.

Inconsistency handling This involves three stages:

- Sample models are used to verify conditions. This set contains syntactically correct models as well as models which violate one or more conditions. The test coverage should cover all present conditions, however it is impossible to verify the absence of errors. This is a caveat we have to take into account. While models could appear as consistent models, they are not always valid models for the transformation engine. In this case we move on to the following stage.

- After identifying and fixing conditions, the resulting correct models are put through the transformation engine. If the transformation fails, this is a signal that despite the conditions, these were not sufficient to prevent a failed transformation. The solution is two-fold: adjust the current conditions to be more strict and add more conditions for a more complete validation of the models.

- After these two stages the next step is evaluation of the result the transformation engine puts out. Even if all conditions are satisfied and the model can
be transformed, the result can be unexpected. In this case several resolution options are available.

- The transformation engine could be the cause, it produces the wrong results. The transformation engine should be fixed. For this purpose additional test data is necessary for evaluation.
- The language could lack expressiveness to capture the intended transformation. Language extensions are necessary.

3.2 Evaluation

With the framework we identify errors in MoTMoT accurately. Other improvements are a better understanding of the different interactions between the components. In the case of MoTMoT we have to separate issues between AndroMDA, MoTMoT and the OCL evaluation engine. Indeed, the framework explicitly separates these Sections and even provides us with the interaction links.

However, the framework is still in its infant stage. While it provides a mapping, true resolution of inconsistencies using information from the framework is not explicitly documented yet.

4 Conclusion

We have shown an implementation of the framework for inconsistency management, developed within MoVES, and its benefits. While the framework cannot be instantiated automatically, the exercise is useful to spot errors in our transformation tool.

The framework is still in its preliminary stages and should be regarded as a prototype. Nonetheless, the instantiations already proved useful as a means of documentation. We have identified that resolution strategies are only tracked manually and investigate the possibility of automatic resolution. The instantiation provides a useful schema for communication and collaboration with different developers and different tools.

The explicit instantiation of the framework for the transformation tool MoTMoT shows the various locations where inconsistencies could occur:

- Inconsistencies identified in the instances of language elements: Violated consistency conditions in AndroMDA show an error message if the validation process fails for the SDM diagrams. This error message, defined by the MoTMoT developer, shows which error occurred and how it should be resolved. Such examples, as mentioned earlier, include missing tagged values, missing type names, ... The user should check the model and identify if the reported error is indeed relevant. If this is the case, the error should be fixed and a subsequent validation will then pass successfully. The end user of MoTMoT is responsible to fix these errors.
- The evaluation tools for the condition language, i.e. the AndroMDA OCL engine, could prove problematic if the tool does not support the complete
OCL specification. In the case of the AndroMDA OCL engine we have indeed noticed failures when OCL constraints involved comparisons to integer numbers. We solved this problem through the use of a helper method which performs the check and returns a boolean value. The AndroMDA developers are responsible to fix these errors, or in the event a fix is not available MoTMoT developers should provide a workaround (if possible).

– Consistency conditions on the modelling language could be too specific or too generic. An example of this problem is the following: an OCL constraint on UML operations demands that every operation has a return type, even though constructors in a class do not have a return type. This consistency condition is too generic and should be replaced by stricter one that excludes constructors. The MoTMoT developers are responsible to fix these errors.

Hence, the instantiation not only allows for a better identification where errors occur, but also who is responsible to provide the necessary corrections and improvements.

References

Comprehensive Support for Evolving Software Cases

Michał Śmiałek
Warsaw University of Technology, Warsaw, Poland
smiałek@iem.pw.edu.pl

Abstract. Software development means producing artifacts at various levels of abstraction: from requirements to code. Every new version or variant of a system introduces changes to all these artifacts starting with requirements. In this paper we propose to form software artifacts into "evolving software cases". Each software case is composed of precisely and automatically mapped models and code. Evolution of software cases consists in introducing changes to requirements models and propagating these changes up to code with comprehensive tool support. The tool detects similarity between new and old requirements and shows places in design and code where rework is necessary. This way, the developers gain instant information on how changes in requirements affect the implementation.

1 Introduction

In various activities within a software project we would like to determine the impact of requirements' changes on the implementation. This is especially important in highly evolutionary (e.g., agile) projects. We would like to know how much time it requires or how costly it would be to amend a system to conform to some extended requirements specification (with e.g., added use cases or user stories). It would be a great saving of resources if we had a tool which would point to specific methods in code that should be adapted (or even adapt some of them automatically). In this paper we propose an approach where the evolution of software is led by evolving requirements written in a language suitable for various application domains (i.e., not a domain specific language from the application domain point of view). Changed requirements instantly and automatically get reflected in the implementation. This is possible through MDA-style automatic transformations from requirements to design and then to code. What is more, this impact is clearly presented to the developers who can quickly find places in code which need adaptation. The main contribution of this paper is to show that the above scenarios are possible for the construction of typical distributed business systems (banking, insurance, production management etc.), with the use of a comprehensive tool suite.

Unfortunately, requirements-based evolution of software is not a frequent subject of research (see [1] for its position within the overall research area and [2] for the current state-of-the-art in requirements evolution). As postulated in [2],
we propose a comprehensive tool infrastructure for requirements evolution. This infrastructure includes automatic transition from domain-independent requirements to design and code. It can be noted that also some previous approaches treat the issues of interplay between requirements and design (see [3] and [4] for important insights). In [5], a process for evolving requirements with separate domain models and associated implementation is presented. Unlike for our approach, automatic transformations and pointers to necessary changes in code are not handled. In [6], theoretical foundations for the evolution of requirements (resulting in their well-formedness) are introduced. Traceability of evolving requirements is also discussed, although again, no automation is introduced. Requirements evolution can also be related to product family construction which is based on requirements variability analysis (see eg. [7]).

This paper follows and significantly extends the process presented in [8] which advocates high levels of automation on the path from requirements to code. We present a requirements specification language where the most detailed specifications can be treated as the first step towards specifying the problem solution (as opposed to the problem definition), which is discussed in [9]. This first step is then translated into component architectures which evolve together with requirements (see [10] for a relevant insight).

It also can be noted, that the current paper tackles some of the research directions presented within [11] (modelling business logic, model interaction), however mostly concentrates on ‘regular evolution’ of models with a forward-engineering approach.

2 Case-Driven Software Evolution (CDSE)

Changes in evolving requirements should be propagated up to code in an automated process supported by a comprehensive tooling framework. The fundamental prerequisites for such a framework are presented in Figure 1. The requirements (R) are formulated through user comprehensible models that cover three aspects: functional requirements, domain definition and non-functional requirements. It is important that the domain definition is precisely separated from the other requirements models. With such a separation of concerns, we allow for evolution of the business domain highly independent from the evolution of the application’s general behaviour (see [12] for an extensive discussion).

![Fig. 1. Tool supported transition from requirements through design to code](image-url)
Automatic generation (denoted by \(\Rightarrow\) arrows, see Fig. 1) of the design blueprints (D) assures that changes in requirements quickly get reflected in the system’s architecture. We identify three aspects within the design (architectural) blueprint: the system’s structure, the application behaviour (sometimes referred to as the application logic) and the domain behaviour (the business logic). The system’s structure is generated from both the functional requirements and the domain definition. Parts of the system are responsible for the application behaviour (mostly: user-system interactions). These are generated from the functional requirements. Other parts are responsible for the domain behaviour (operations on domain elements, or: data processing). These are generated from the domain definition.

Further on, code (C) is updated accordingly, through consecutive automatic transformation from design. In this paper we will not treat in more detail the D to C transformations which have a vast literature. It has to be noted, though, that with the current state-of-art, the whole process cannot be automated fully. Important aspects of the requirements specification have to be propagated to code by the designers/programmers manually (denoted by the \(\rightarrow\) arrows).

Automatic transformations are controlled (denoted by \(\rightarrow\) arrows) through appropriate transformation specifications (R to D and D to C). These specifications capture important design decisions like the architectural style or design patterns used. They thus add the technological aspect to the requirements models. It can be noted that it is the non-functional requirements that get reflected in the transformation design and the choice of the technological framework (like EJB vs. .NET). This choice of the framework depends on various quality requirements (like maintainability or portability), and currently is a manual process based on the experience of the system architects. The choice of patterns and styles for the target design models also depends on the non-functional aspects. Generally, the functionality of any system can be implemented within various design styles or patterns (like model-view-controller, multi-tier, service-oriented and so on). However, if we analyse the non-functional constraints (like performance, interoperability), only some are suitable and will allow for passing of appropriate tests (stress tests, etc.). This analysis should be done by the transformation specifiers and again is a manual process based on their experience.

In this paper we do not treat the issues of transformation evolution. We thus fix the non-functional requirements and the technological framework (denoted with ‘=’ in Fig. 1). On the other hand, we allow for changes in functional requirements and domain definition (denoted with ‘\(\neq\)’).

We will now briefly introduce a transformation from R to D complying to a wide range of non-functional constraints (see [13] for details). It is illustrated in Figure 2, where the requirements model is composed of use cases and domain notions. The use cases have model-based scenarios. The domain notions contain verb-based domain statements with appropriate specifications of their logic. The target design model contains three layers: presentation (not shown for brevity), application logic and domain logic. The technology framework is component-based, and thus the component model is used. Every use case is transformed
into an interface of a component in the application logic layer. Every domain
tonotion is transformed into an interface of the business logic layer. Appropriate
operations of the interfaces are generated from scenario steps and domain
statements. What is not shown in the figure are the models of the system’s behaviour.
Appropriate interactions between the layers can be presented in sequence dia-
grams. These diagrams represent the application behaviour and are generated
from scenarios (sequences of scenario steps). This is illustrated in an example in
the next section. From the sequence diagrams, the methods for the application
logic layer interfaces can be generated in code. The methods for the domain
logic layer interfaces have to be derived from the domain statement descriptions
(this is the ‘manual’ part of the process). It has to be noted that the important
prerequisite for the transformation is that the scenario steps are precisely linked
(hyperlinked) to the domain statements. Only this ensures proper ‘wiring’ of the
resulting component and interaction models.

With the above transformations we can also generate detailed mapping (trace-
ability) information between three main artifacts (R, D, C). With these mappings
we can combine R, D, C into a single artifact which we will call a ‘software case’.
Software cases can be further combined into ‘software case sets’. Each software
case in a set reflects a single (possibly partial) version or variant of a system.
In an iterative process, software cases in a specific set reflect consecutive builds
of the considered software product. The evolution of software systems based on
software cases is illustrated in Figure 3. We first take Software Case 1 (SC1) from
the software case set. This SC is reworked by changing R1 (adding $\delta R_2$). This results in R2 from which we generate D2 and (partially) C2. With an appropriate tool we can now visualise places in C1 which can be retained into C2. Other pieces of code (operations generated from R2) have to be written or re-written. The tool helps us by pointing to places in C1 which can be taken for adaptation.

Another evolution scenario can start with a completely new requirements specification. With this (probably incomplete) specification we can query ($Q_{R4}$) the software case repository. The tool finds the most relevant software case in a software case set (in fact: in a global software case set containing software cases for different products). It also shows the differences between R1 and $Q_{R4}$ ($\delta R_4$). Knowing this delta facilitates reuse of R1 and preparation of R4. It also points to places in code for rework as in the previous scenario.
3 Case study

In order to show that the above CDSE schema is valid we will present a case study example. This case study includes three steps in the evolution of a simple software system in the Campus Management domain. We use a comprehensive tool (ReDSeeDS Engine [14]) with which software cases can be created, queried for, and visualised for differences. The tool allows for creating requirements models in a special purpose Requirements Specification Language [15]. The functional requirements are based on the use case model with precise syntax for the scenarios and associated domain elements (see [16]). The first step is denoted as R1 in Figure 4. It contains two use cases (only the basic path scenarios shown) and two domain elements (‘course’ and ‘course list’). The domain model contains all the verb phrases used within the scenario sentences. It is created semi-automatically when writing scenarios.

In the next two steps, new elements are added to the requirements models ($\delta_{R2}$ and $\delta_{R3}$). The second step adds filtering capability to the ‘Show extended course list’ use case. The third step further extends this use case and adds three additional use cases. Together with extending the scenarios, the domain model is updated with new notions and statements (verb phrases).

While extending the requirements, also the design models are extended. This is shown in Figures 5-6. These figures contain complete structural blueprints for the three presented steps. These models are generated fully automatically from the presented requirements. Comparison of Figures 5-6 with 4 shows how the transformation rules from Fig. 2 were applied.
The difference shown in Figure 5 is easy to determine. Also, the influence of changed requirements can be quickly determined. However, this is not so evident for the changes in the third step which are more substantial. This would be even less evident for larger systems with tens or hundreds of components. Thus, the transformation system also generates diagrams showing interactions for individual use cases (see Figures 7 and 8). These diagrams allow for determining the impact of changes in scenarios of an individual use case. It can be noted that the interaction diagrams determine the contents of methods at the application logic layer. For instance, the 'entersCourseFilter()' method (see Fig. 8) consists of two calls to other interfaces, one of them being conditional. In fact, the transformation engine could generate this code automatically, and only minor manual updates would be necessary.

Usually, the requirements for a new version of a system change in many places. The changes might mean also deleting some functionality. It might be the case, that the new version is more similar to some older version (perhaps in only some places). Thus, we would need a mechanism which would show clearly the differences between various software cases in a software case set. In the ReDSeeDS Engine, such mechanisms constitute a combination from information retrieval, structure-based and semantic-based similarities (see [17] for details).
The results of an appropriate query within our case study example is shown in Figure 9. There, the third version of requirements is compared with the previous two versions. Actually, the engine seeks within the whole repository which contains also other software case sets (for other systems). The two previous versions were found as most similar (the only ones above the threshold of 0.3). The engine shows also the similarity between individual use cases. It can be noted that by using the engine we can also find recurring functionality within the software case. For instance, we can reuse significant part of the application logic code of the old ‘Add new course’ use case when developing for the new ‘Change course data’ use case.

The final action after determining similarities is to check for the code to be reworked. We can also use the tool to do that. For every use case under consideration, the tool can show us traces to individual operations of interfaces that are used to implement this use case. This way, the developers gain instant information on the methods for these interface operations which need to be changed or can be reused from the previous version. Figure 10 shows traces for two selected use cases in a ‘2-tree view’. One of the trees allows for selecting use cases to trace from. The other tree shows all the elements of the component model which are affected. The tool can also include one more tree (‘3-tree view’) so that also the most detailed code model can be traced into (not discussed here for brevity).
Fig. 9. Showing a $\delta_R$ through similarity values

Fig. 10. Showing traces from requirements to operations (code)

4 Conclusion

The CDSE approach, together with the ReDSeeDS Engine constitute a powerful mean of support for the software developers. The main benefit is that the developers (and project managers) can see the consequences of changes to requirements before they are coded. Thus, they can see a ‘diff’ for even a yet not existing code right after the requirements are ready. The differences are illustrated by instantly showing places in the design models where code rework is needed. Another important benefit is that the system refactors the design automatically based on the changed scenarios and domain notions. The developers have clear pointers, as to which methods should be changed (extended, combined, split, etc.). This is clearly shown by comparing visually with the previous version.

A software case repository can hold many software cases for various domains. Thus, after conducting several projects with CDSE, a development organisation obtains a powerful repository of reusable artifacts. These artifacts can be reused by building new versions, but also a wider (cross-domain, cross-system) reuse is possible. This can be done by merging use cases and associated ‘trace-based slices’ (containing components, interfaces) from different software cases. It has
to be noted that software cases offer specific solutions to specific problems. This
is different to software product line and pattern approaches where the problem
and its solution are generalised by default. Software case repositories become
general through being populated with more and more software cases covering a
growing range of possible applications.

Acknowledgments. This work is partially funded by the EU: Requirements-
Driven Software Development System (ReDSeeDS) (contract no. IST-2006-33596
under 6FP).

References
2. Ernst, N.A., Mylopoulos, J., Wang, Y.: Requirements evolution and what (re-
3. Nuseibeh, B.: Weaving together requirements and architectures. IEEE Computer
34(3) (2001) 115–117
5. Barber, K.S., Graser, T.J., Grisham, P.S., Jernigan, S.R.: Requirements evolution
and reuse using the systems engineering process activities (SEPA). Australian
6. Zowghi, D., Gervasi, V.: On the interplay between consistency, completeness, and
correctness in requirements evolution. Information and Software Technology 45
(2003) 993–1009
7. Moon, M., Yeom, K., Chae, H.S.: An approach to developing domain requirements
as a core asset based on commonality and variability analysis in a product line.
8. Śmiałek, M.: From user stories to code in one day? Lecture Notes in Computer
9. de Boer, R.C., van Vliet, H.: On the similarity between requirements and archi-
Notes in Computer Science 2817 (2003) 152–163
of Systems Informatics PSI09, Novosibirsk, Russia (2009)
Publishing House of the Warsaw University of Technology (2007)
16. Śmiałek, M., Bojarski, J., Nowakowski, et al.: Complementary use case scenario
representations based on domain vocabularies. LNCS 4735 (2007) 544–558
17. Welter, K., Krebs, T., Hotz, L.: A combined similarity measure for determining
similarity of model-based and descriptive requirements. In: Proc. AISEW Work-
shop at the ECAI 2008. (2008)
Practical Case Study of MDD Infusion in an SME: Final Results

Regis Vogel, regis_vogel@yahoo.com

Abstract. This paper presents the results of an experiment conducted in a systems integration company investigating the impact of MDD infusion. For systems integration companies, the ability to quickly evolve and assimilate technological advances is a critical asset and a differentiation factor. Our approach takes into account the practical aspects of such assimilation during project activities, including the influence of the motivation and beliefs of the technical developers on the success of the technological change. Another practical aspect we’ve analyzed is the process through which organizations generate value from their intellectual assets. In this paper we delineate the industrial relevance of the results of this MDD study, defined here as being proximal to real, applied experiences in industrial settings that industrial actors (project managers, developers, etc.) can relate to, draw conclusions from and translate into action. Since the experimental design and partial findings related to this MDD study have been presented in previous publications, here we limit the focus to the final results and conclusions.

Index Terms—Model Driven Development, Model Driven Architecture, Software Engineering, Software Development, Software Maintenance, System Integration, Case Study, Survey Learning Curve.

I. Introduction

Though it could be expected that MDD’s high potential for positive impact in production environments/settings would be easily perceived, there is currently little evidence that this is, in fact, the case. The true potential for positive impact is moderated by practical factors such as the organizational actors’ skills, motivations, beliefs and interrelations. The roles that these factors play in successful infusion are as important as that of the scientific quality of the processes and methodologies being infused. The purpose of our approach, therefore, is to identify and qualify the impacts of infusing MDD in an SME (Small to Medium-sized Enterprise) and to get a “close to real life” result.

Towards this end, we designed and ran an experiment that included three studies, various surveys and participant feedback. The research design of the experiment and partial results have been presented in previous papers and workshops [2][3][4][5]. The complete description of the experiment, including tool chain detail, studies, surveys, etc., can be found in “D5.3-2 Enabler ROI, Assessment, and Feedback” [6]. The objective of this paper is to present the aggregate final results and conclusions from the overall experiment. In Section II we provide a description of the experiment, including an overview of the tools and processes used. In Section III we present the results of the three different studies, and in Section IV we review our conclusions.

II. Description of the Experiment

Right from the start of the project, we identified that infusing MDD in a company would have at least three impacts on organizational structure:

• A change in the tools people are accustomed to using;
• An adaptation of the actual development process and the actors’ roles;
• A change in “mind-set” from a code-centric approach to a model-centric one.

Therefore, our experimental design responds to these questions about infusing MDD in an SME[2]. The human factor plays a dominant role in the success of any organizational change, including change of a technical nature. If a technical change is to achieve success in improving software development
performance, this factor must be taken into account. The experiment we conducted allows assessment of the human factor through three studies:

- **Software Development Study**: Development of a complete module using both traditional and MDD methods.
- **Software Maintenance Study**: Once module development is complete (first study), functionalities are added to user maintenance using both traditional and MDD development methods.
- **MDD Education Study**: Educational actions focused on model-driven concepts, tools, and methodologies have been conducted simultaneously with the above-mentioned studies.

It should be noted that in order to perform an MDD maintenance study, creation of an MDD development environment is required. Having a side-by-side comparison of the costs of maintenance after the initial development environment has been created (during the Development Study) allows us to isolate the maintenance costs as they exist separately from the initial development. For example, if it takes twice as long to develop the initial MDD environment and only 50% of this effort/cost is recouped during the maintenance stage, one conclusion might be that the ROI calculation for the MDD environment favors long-life applications or those that require frequent updates.

To help us capture the perceived value of MDD for different job functions and technical areas and its perceived impact on the learning curve, as well as to provide insight into the subjects’ views on advantages, constraints, and concerns about MDD, we developed four distinct surveys (skill, satisfaction, improvement, session feedback). The surveys were taken before, during and after the studies. Detailed descriptions of the studies and the surveys can be found in [5] and [6]. Figure 1 presents the schedule of the Software Development, Software Maintenance and MDD Education studies that compose this experiment. Groups T and V from the Software Development Study consisted of one subject each. Subjects have similar skill levels, and little work experience or business background. While no statistical data can be gleaned from this case study, it is indicative of expected results and points to the need for large-scale studies with similar structures to be performed.

Developers in groups Y and Z (four people total) exhibited diverse levels of experience and business knowledge. Across the two groups, however, the subjects were well matched, allowing direct comparison between the two groups. These subjects came from the pilot education group. They are from different departments of the company and were selected to participate in the project because they would accurately report the expectations of their departments. Their skill levels are presented in [6].

![Fig. 1. Studies schedule](image)

The MDD education study has been performed with a non-random group, identified as the pilot educational group, including subjects with varying technical skills and job functions (e.g., project managers, software developers, software testers). Depending of the session the group size varied from 5 to 12 individual.
1. Experiment context

The research is being conducted as part of Enabler’s participation in the ModelWare [1] IST project. Enabler is an SME specialized in the creation and integration of IT solutions for retailers. This software development/integration domain is characterized by short lifecycles, rapidly changing requirements and heterogeneous implementation platforms. One of Enabler’s characteristics is geographic dispersion of teams, including its headquarters in Portugal, a software factory in Brazil, offices in Portugal, the United Kingdom, Germany, France, Brazil, Italy and Spain, and other operations throughout the Americas, Europe and Asia. In addition to its core services, which include systems integration, quality assurance, consultancy and applications, and services maintenance (ASM), Enabler is also developing a line of software products/frameworks. The experiment underlying this paper is based on Enabler’s “Pricing Framework”, an application designed to help the retailer determine the price of a product using built-in, configurable business rules. It’s developed in Java with a three-tier application architecture: a rich Java client, a business application supported by JBoss application server and an Oracle database.

2. MDD Process

Enabler’s current “Pricing Framework” software development methodology is solution integration, which consists of waterfall and ISO 9001 certified process using sign-off.[3] It is supported by MS-Word templates for contractual and development documentation, but it is not supported by any specific tool or integrated toolset; validation and verification is done manually. Each phase of the life cycle uses distinct tools and methods specific to its goals. The process is heavily manual, however, one of Enabler’s advantages is that its consultants (even within the team of Java programmers) are specialists in the retail domain. This industry knowledge supports the process of understanding, correcting, and extrapolating from any inconsistent documentation provided by the analysis and design phase.

Basing our MDD process on the underlying approach of “evolution not revolution” for the conservative retail domain, we have minimized the impact on the different stakeholders whenever possible. In fact, the MDD process itself is not very different from the traditional one described above, is composed of the same phases (requirements gathering, analysis, design, development and testing), and produces the same output documents. However, the characteristics of the artifacts that are created during the development process, the distribution of effort among project phases, and the process for generating these documents are all very different. Due to the existing level of UML knowledge within the company, the Modelware project and the community, we selected UML2.0 and UML profiles rather than DSL to support our MDD methodology.

3. MDD Tool Chain for Enabler

One of our main objectives when selecting the different tools and creating the tool chain was providing an integrated environment to support the MDD process, to hide and capture complexity and to avoid context-switching between tools. Doing so will help users consistently focus on their work at the model level. Cost, identified from the beginning of the project [2] as one of the key constraints for successful adoption, required that we select tools that are compatible with an SME’s budget.

Since no single tool provided the different services needed, we integrated several of them. Figure 2 reflects the tool chain used for the maintenance experiment.

The following list reflects the components of the integrated tool chain:

− **Modeler**: Rational Software Architect (RSA) is used to create profiles, manipulate our UML2.0 models and aggregate/present all the other tool services plugged into the Eclipse platform. Running on Eclipse the tools have access to the memory representation of the models, which solves the problem of model/metamodel communication/exportation that confronted the ModelBus. So RSA is in fact our main entrance into the MDD tool chain since almost all user activities must start from this tool. As an example, we implemented the orchestration tools using the pluglet feature provided by RSA.

− **OCL- OSLO**: This tool is used for validation of the model and for quality control. We believe that constraints on quality control do not necessarily need to be implemented in the metamodel, but rather can be done in an external file because their life-cycle may be different from that of the meta model. Further, the quality level rules may differ for different projects, but the meta model remains the same.
Therefore, we developed an RSA pluglet for quality validation using two files containing OCL Rules: one file for “Warnings” and another for “Errors”. The pluglet allows the user to navigate through any errors and link directly to the element that doesn’t respect the rule, optimizing the validation and correction process. While RSA has an OCL engine and an OCL syntax parser, its engine did not provide the OSLO service of running as a batch with a list of constraints for validation.

Fig. 2. MDD tool chain

- **Requirement tool**: Mantis is a bug management tool that we tailored for requirement management. It is not integrated on ModelBus, however we developed an export to a UML2.0 model with our specific requirements profile. This allows us to manipulate the requirements within MDD tool chain and therefore use its services, such as the UML editor, OCL, transformation and traceability tools.

- **Model-to-Text: MOFScript**:
  - Documentation generation: Within the Enabler experiment, model-to-text transformations are used to generate documentation based on the company’s standard templates. One major requirement for the generated documentation is to follow this template as far as possible. Besides maintaining the template, the problem of merging the handcrafted part and the automatically generated part into the document had to be solved. We used Open Office as a target. The ODF format is cleaner and easier to manage than RTF from Microsoft. Additionally, the concept of a Master Document allows us to separate the delivery document into manually edited and automatically created parts.
  - Code generation: In addition to documentation, model-to-text code generation is used to generate Java as well as the configuration files needed in the targeted platform. At the end of the experiment Java and all configuration files were generated.

- **Model-to-Model: ATL**: The model-to-model tool selected is ATL[9]. We use it to generate analyses to design models, and find it especially useful for the GUI (Graphical User Interface), with the profile for GUI. However, the limitations and bugs of the tool when managing models with profiles required us to shift focus to other tools such as model-to-text. Additionally, we experienced problems during the merging that caused loss of information. Therefore we decided that for the Maintenance experiment, the models will already be present and will only be updated manually. We hoped to revisit this before the end of the project but were not able to do so.

- **Orchestration tool**: This tool uses the RSA extension mechanism (pluglets), and invokes the transformation processes. By “transformation process” we mean first calling the OCL to validate the quality of the model, then invoking the transformation. Doing so we hide the complexity from the user, since he doesn’t need to know which transformation file to call. Additionally, this provides information about the quality/validity of his model. This user-friendly integration was one of the main objectives of the MDD tool chain.
- **Traceability tool:** Imbus developed the traceability tool according to our requirements during Modelware project. It is used to maintain links (dependency or traceability) between artifacts created/generated during the MDD process and does so independently of the tools that produce them. This feature allows us to perform queries to get, for example, the impact analysis of a change, or simply the ability to identify whether a requirement is described as a use case. The traceability client integrated in RSA allows us to trace any kind of artifact that is present in the RSA workspace.

- **SVN Subversion:** This is the repository management tool for the project. In the Enabler MDD process all artifacts used and generated must exist in the RSA workspace and in the SVN repository. SVN integration with RSA and Subclipse supports the merging of models and version management. The Subclipse plug-in for Eclipse provides us with graphical and textual views of model differences between model versions. We use this tool to merge manual modifications of a model and wished to use it for generated modifications as well. The decision to use SVN as repository instead of a MOF repository such as the one from Adaptive was made because the current functionality obtained with the SVN/RSA solution was sufficient for our experiment. License price was also a consideration in the decision.

- **Maven 2.0** is used to present the project information (tools, participants, etc…) as well as to publish the generated documentation to the project intranet each time a release is made. The objective is to have Maven as a central tool for managing releases, information about the project and continuous integration.

### III. Studies

1. **Software Development Study**

The objective of the software development study is to provide industrial relevance to the experiment. The research design is a case study. The non-random group V consists of one subject developing a software component (“user maintenance”) using MDD. Group T consists of one subject developing the same component using the traditional method. Both subjects are junior developers. This design, although not valid statistically, is indicative of the impact of MDD adoption using subjects with similar profiles, as well as the cost of the development with both methods.

To gradually introduce MDD technology and knowledge to the participant, as well as to facilitate a rolling introduction to the tools depending on their availability/stability, we planned this study in various iterations. The details of each iteration (work plan, results, challenges, etc) can be found in [6] p.32. The comparison information between traditional development and MDD has been extracted based on the time to complete the iteration “MDD Module Development” where a full MDD module was developed. This is an indicative data point useful for tempering the results of the MDD participants.

This study was also interesting when comparing the MDD feedback since it was based on the Development Study, which comprised development of a full module. In contrast, the Maintenance Study was only a short modification on an already-existing environment.

Two skills surveys were completed [6]. Participants took the first at the beginning of the project, and the second at the end. The result of the first survey, with a 10% correct response rate, shows some MDD knowledge acquired during the model creation task for design validation purposes. The end result, with a 100% correct response rate, shows that all identified basic MDD knowledge has been acquired. In fact, although the participant did not take a survey with enough complexity to measure all his knowledge acquisition, we can deduce that his understanding of MDD, UML2.0, and the different techniques and platforms used in the project increased significantly, reaching a level that made him comfortable working in that area. Even so, additional work would be needed to become proficient using ATL tools [9].

As previously noted, this is a case study and is not statistically reliable due to the number of participants (only two with no cross over), stability problems, issues with communication among tools, etc. However, as a case study it is sufficiently representative to reflect the key issues in MDD vs. traditional development comparisons.

Globally, the MDD implementation required 27% more time than the traditional one. However taking into account the time spent overcoming tool chain issues (17%) we have a final difference of 6% in favor of the traditional method. While the overall time spent to implement that module is similar between methods, the following observations should be noted:
When compared to traditional development, we notice that MDD requires a transfer of effort from development to analysis and specification. A significant amount of work is required in the initial definition of a system in order to fully explore the potential of the MDD. When this initial work is done, the extension changes are faster.

The time spent in these first phases (Analysis and Design) are greater than expected, which can be attributed to the fact that the models had to be created by hand since model-to-model generation from one level to the other wasn’t available.

2. Software Maintenance Study

The scope of this study is to apply change requests to the User Maintenance module of the Pricing Framework. Of comparable complexity, the two change requests under evaluation are the inclusion of the following concepts:

- **Cutoff date**: After a certain date the user can no longer log into the system
- **Password validity**: The user password must be regularly changed.

The MDD development environment used in this study, as described above, was available on each computer. Similarly, within the traditional environment all tools, the CVS repository and database server were available to recreate the traditional working environment for the Pricing Framework project. The computers were identical as they were installed from an image of a master computer.

We divided the subjects in two groups, Y and Z, and used the Crossover design: both groups were subject to two treatments: “use of model-driven development” and “use of traditional development method” in two periods. These were performed sequentially, with each group performing one change request using MDD and the other using the traditional approach, as reflected in Table 1. Since the comparison was carried out for each individual, the comparison of treatments was not contaminated by the variability between subjects. This design also eliminates the practice effect (group Y has the practice effect added to the control treatment while group Z has the practice effect added to the study treatment) leaving only the treatment effect to be evaluated in the study.

The experiment took place in the Enabler office on 06-07/06/2006 (see Table 1). Four people involved in the MDD training study participated in this experiment. The participants’ positions within Enabler, their skill levels and experience can be found in [6].

<table>
<thead>
<tr>
<th></th>
<th>Add a cutoff date to the user</th>
<th>Add a limit to the validity of the password</th>
</tr>
</thead>
<tbody>
<tr>
<td>Groups Y (X1, X2)</td>
<td>MDD: 06/06/2006</td>
<td>Traditional: 07/06/2006</td>
</tr>
<tr>
<td>Groups Z (X3, X4)</td>
<td>Traditional: 06/06/2006</td>
<td>MDD: 07/06/2006</td>
</tr>
</tbody>
</table>

Table 1. Group Crossover planning

Since the study was performed both with and without MDA tests, its effectiveness in providing fast solutions for requirement changes and bug fixing, which is where most development effort is usually spent, is tested. This study also shows the results with the current methodology. The results presented in Table 2 reflect the comparative information. Please refer to [6] p.37 for more detail on each part.

Each participant registered the tasks, task descriptions and the time dedicated to each task in an Excel file. Timers were available on each computer to allow the participant to monitor his time. This information allows us to calculate the downtime of the platform and the tasks that the participant performed, the task order and the participants’ assessment of time spent. The compilation of this Excel data is available below.

We didn’t provide a task template for the Excel registration file since doing so would have specified an approach to the work, which was contrary to our objective of observing the participants’ working methods.

Because the duration of the experiment was limited, we used an “achievement order” to ensure homogeneous task completion. Each achievement was specified by a list of tasks each participant was required to perform in order for the change to be considered implemented. While performing the change management, the participants were required to update / generate the following “end” artifacts:

Results

Table 2 presents the comparison of achievements between the MDD and traditional method. “tra” represent the % of achievement with traditional methodology, “MDD” represent the % of achievement with MDD methodology, “Diff” is the difference between the two. At 81%, the average overall achievement value of the MDD experiment was substantially higher than that of the traditional experiment (24%).

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Seniors</th>
<th>Junior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement</td>
<td>TRA: 60% MDD: 65%</td>
<td>Diff: 5%</td>
</tr>
<tr>
<td>Analysis</td>
<td>TRA: 20% MDD: 70%</td>
<td>Diff: 50%</td>
</tr>
<tr>
<td>Design</td>
<td>TRA: 0% MDD: 90%</td>
<td>Diff: 90%</td>
</tr>
<tr>
<td>Development</td>
<td>TRA: 85% MDD: 100%</td>
<td>Diff: 15%</td>
</tr>
<tr>
<td>Total</td>
<td>TRA: 42% MDD: 92%</td>
<td>Diff: 50%</td>
</tr>
</tbody>
</table>

Table 2. Maintenance Study Achievement Comparison Table

It is important to note that with the traditional methodology the senior participants’ achievement for development was more than 80% (94% on average). X3 with 100% even provides the complete working implementation of the change requested. This is mainly due to the fact that their experience and familiarity working with code made this their first recourse, before sourcing other options such as documentation, etc. Making a breakpoint in the code allowed them to see what do to, and constituted their entry point for the change implementation. However their overall achievement at 41% is low due to the poor results in the other required tasks (for analysis and design, none of the required documentation was updated). With an average achievement of 8%, the junior participants’ results show that they were not able to master the complexity of the project platform, and that they were not able to navigate large quantity of project documentation available.

With MDD, the achievement percentage for each participant increased on average by 57%, despite the reduced duration of the MDD experiment and the tool chain stability problem. Considering achievement averages, the best results were in the MDD environment where participants actually completed tasks instead of being lost in reading or looking for difficult-to-find or non-existent information. We note that the discrepancy in achievement levels favouring the MDD environment over the traditional environment would probably be significantly more marked had the developers been subjected to an unknown “traditional” environment rather than Enabler’s standard environment, with which they were already very familiar having worked nearly their entire career at Enabler.

The achievement difference is greater for the junior participants (X2 and X4), with 64%, compared to 51% (X1 and X3) for the experienced participants. While the difference between these two groups was 33% in the traditional development environment, it was reduced to 20% with MDD.

With an increase of 123% the achievement for the senior developers is gained in the requirements, analysis and design tasks. This improvement is largely due to the traceability and automatic generation that helped update information that they didn’t find or didn’t complete in the traditional process.

The largest increase obtained with MDD is for the junior developers. The fact that there was an 8-fold (847%) increase between traditional and MDD environments suggests that the MDD platform/process hid the complexity and allowed the developers to focus only on the business implementation.

While the data is not presented, here we found interesting that there was much less variation in the time duration recorded by experienced vs. inexperienced participants within the MDD environment. Similar to the interpretation mentioned above regarding the difference in total achievement in the two environments,
we posit that the reduced variation between experienced and inexperienced groups in the MDD environment may be due to the fact that all participants were new to MDD, thus reducing learning-curve effects based on development experience, both general and specific to Enabler’s methodology.

Although the platform was not stable, (26% of the experiment’s run-time was dedicated to tool chain stability problems) the participants appreciated the tool chain. This was especially true regarding the seamless integration of the traceability tools, the validation tools and the repository within RSA. This integration gave a lot of power as it increased visibility over the impact of changes and avoided environment switch.

Figure 5 presents the outcome of the quality survey gathered after the experiment. This graph shows the response to the following questions:

- **Reliability**: Do you trust the MDD process language and tools you use?
- **Efficiency**: Do the MDD process language and tools help you do your job faster?
- **Integrity**: Do the MDD process, language and tools alter the information you entered?
- **Easy to use**: Are the MDD process, language and tools easy to use?
- **Easy to maintain**: Are the artifacts that you manipulate in the MDD process easy to update?
- **Easy to test**: Are the artifacts you are manipulating easy to validate/test?
- **Reusability**: How do you see the reusability of your work?

The average of 5.7 over 9 (63%) is relatively low. We can see that the two lowest data points are Reliability and Integrity. This is not surprising, considering the stability issues we faced during the experiment.

The best average is set at 6.4 by X2 (less than one year of experience) who gives equilibrated ratings to the different areas, while the second best average at 6.2 is provided by X1. X1 is the most experienced participant of the group (7 years), and is the participant with the highest achievement. We can see that his Reliability score is his lowest value (4), Efficiency (5), the second lowest one, which is probably due to the first result. Looking at the other scores (respectively 6, 7, 7, 7) we can conclude that X1 appreciated the MDD tool usage. X4’s (1 year of experience) highest value is for Efficiency (8), which corresponds with his enthusiasm for the traceability feature that was guiding him through the part that needed to be modified. This contributed to a higher confidence level regarding his work, and gave him the greatest achievement increase of all the participants. X3’s (5 years’ experience) relatively negative evaluation was, according to his final review, due to the instability of the platform, which generated frustration as he knew that if the tools and the platform had been stable enough, much more could have been done.

The Education Study is part of an Education Plan that was developed to introduce MDA within the company and prepare team members for their initial use of MDA. To capture the learning curve and
dissemination, we performed an evaluation survey of the education actions. All feedback has been used as an input for subsequent education actions in order to increase the efficiency of the training sessions and to raise trainee satisfaction. Feedback on the “MDA Education Series” has been incorporated, whenever possible, in the subsequent editions. To facilitate the introduction of the technology, an MDA course with Enabler-specific content has been designed (MDD framework). Additionally, a forum and a news group have been created to give visibility of the project to the rest of the company. We tailored the classroom training and lectures with specific content based on both the profile of the attendees and the objective of the course. Due to the availability constraints of the participants, we packaged the lectures into three sessions. Below we discuss the result of the surveys given at the end of each session.

The overall session feedback survey [6], with a participant average of 75%, reflects high interest and positive reviews of the training. X4 rated five criteria at the maximum score of 4, reflecting a good match between his current work and the information/knowledge he obtained during the experiment. X5’s score is low due to her rating on the MDD Framework. Motivation and Interest, at 85%, shows that our goal of winning “hearts and minds” has been achieved. And “Relevance to Your Job”, with 77%, shows that participants do make the link between the training and their work.

For the first session, “Perceived Ease of Use” and “Perceived Compatibility” were among the lowest values. This is to be expected when change is introduced to a group, and particularly so for a paradigm change as dramatic as the one introduced in this “code-to-model” experiment.

The second session was the most positive of the three sessions. Results improved on all evaluation metrics, and particularly so for “Perceived Ease of Use”, “Perceived Compatibility” and “Training Quality”. These results are reflective of the fact that the participants in this session understood and actually executed the MDD process with a concrete example. The values for this group appear to show that this training approach made them more secure about how to use MDD, and that MDD may be more compatible with their working style and methods than they expected. “Perceived Usefulness” is the only metric that decreased (4%) for this session. One hypothesis is that participants’ expectations in this area were raised due to the its high rating for the previous session.

Session 3’s results were less positive on all metrics than the former two sessions. One impediment to this group’s success was platform stability, which added an element of stress for these participants as they attempted to deliver results under a time constraint. The “Involvement Satisfaction” value, the only metric showing improvement, reflects the commitment of the participants to this experiment and to continued promotion of MDD. This result, in conjunction with the decreasing value of “Communication Satisfaction”, (the first in the experiment) can be interpreted as showing that participants think more should be done to promote the project.

The evolution of responses in the Satisfaction Survey reflects improvement on nearly all metrics as compared to the first session. At the time that this survey was taken, participants had received more information and therefore had deeper knowledge of the process and tools. At this point they could better perceive its impact, and were less concerned about the change. The only headers that evolved negatively were the “Future Use Intention” and “Perceived Usefulness”, which is probably accounted for in some part by the platform stability problems that arose during the experiment.

The Skills Survey was taken after each session, and shows that the majority of the knowledge has been assimilated but more time and practice are needed for complete mastery.

The Satisfaction Survey, completed during the training sessions and after the Maintenance experiment, provides sufficient material to evaluate the resistance level. The valuation of the most and the second-most important features of the survey, show that even if the feature itself changed for each training session, it is generally in the “Perceived Usefulness” section (75%). For the least important feature we obtained a 75% score for “Perceived Ease of Use” and 25% in “Perceived Compatibility”. Those values show that participants think that MDD will help them in their work and that they are ready to make the change despite the extra effort and the change in their current working methods. Training was tailored to the participants’ specific development process, using specific domain examples. We have seen that using this strategy improved the effectiveness of the training and additionally helped sway the “hearts and minds” of the participants. The result of the non tailored (RSA) training, which obtained a score of 50% for the “Relevance to My Job” variable, versus 83% for the entire session shows the importance of the design/content of the training on the potential for change to be accepted and integrated by developers.

The Improvement Survey was distributed at the end of the experiment. The conclusions derived from this survey are presented in the “Software Maintenance Study” section.
IV. Conclusions:

1. Validation of Hypotheses

In the beginning of the project Enabler identified the possible benefits of using MDD [6] P.14, and estimated that a 20% gain in productivity could be achieved as an objective measure of the achievement. The results of the experiments undertaken and presented in the previous sections of this document and summarized in the above conclusions proved most of them and have quantified a productivity gain of 57% for the Maintenance projects.

Three hypotheses were defined in the design of the experiment [2]:
- H1: Components developed with MDD demonstrate higher performance (in software development and maintenance) than components developed using traditional methods
- H2: The tools’ maturity will moderate the effect of MDD adoption on the software development performance
- H3: The perceived value of MDD moderates the effect of MDD adoption on the software development performance.

Compared to traditional development methods, MDD implies a transfer of effort from development to analysis and specification. A significant amount of work is required in the initial definition of a system in order to fully explore the potential of the MDD features such as meta-model definition, automatic code and document generation, traceability and automatic testing. This was shown in the results of the Software Development Study where the extra costs in these areas meant that MDD productivity could actually be identical or worse than the traditional approach. Indeed, MDD delivers more value in software maintenance than in software development since software maintenance usually focuses on further developing existing software. In this experiment, the system specification and infrastructure setup (which are the phases requiring the most effort in MDD) were already completed. Therefore, the Software Maintenance Study has corroborated the H1 hypothesis. The tools stability there is not relevant since we are evaluating achievement. The results obtained (-27%) during the Software Development Study do not corroborate the hypothesis. However, if we dissociate the time spent on stability issues (17%) by removing it from productivity, the difference with traditional approach is then -10%. The design of the experiment, with only two participants in different contexts, suggests that this value may be accounted for by many other factors. Additionally, the higher start-up costs for MDD should be considered an investment dependent on the amount of maintenance that is foreseen for the project. The effort required to perform the setup could decrease with a team experienced in MDD and more fully-featured tools.

The H2 hypothesis has been verified in both studies. With 17% for the software development study and 26% for software maintenance, the stability issue definitely moderates the effect of MDD adoption on the software development performance. Experienced participants were hesitant to endorse this methodology within the production environment. A general comment by one of the participants reflects the feeling of this study: “On the end of the first day it was clearly visible to us that, using our current method, we lost a lot of time in trying to figure out the functionality of an application, the impacts of our changes in it and getting the broad overview of the all application and necessary changes needed to fulfill the client’s request. On the end of the second day my thoughts were that, with the new method, the time to get the broad overview, document changes, etc… would be less than on the first day and less hard also, but the tools available felt more beta-quality (working and not working at all sometimes :) ) than the ones used on the first day (more a feeling of production quality). Today’s Enabler method is not MDD but it seeks answers to its defects that are leading it, in my opinion, to MDD. But people only feel comfortable on solid ground. In my mind Modelware is aiming to provide the right tools and ideas to give a solid ground felling but it still needs time to mature from a beta feel to a production feel.”

The H3 hypothesis has been neither verified nor rejected during the MDD Education Study. Study results show that by creating specific training adapted to knowledge and domain, and by providing concrete examples, we avoid that effect in the overall training. However it is noticeable that we obtained that particular effect after one specific course that was not tailored to the audience.
2. General Conclusions

The introduction of MDD appears to have directly and positively impacted the development process in the following ways:

- **Simplification and Traceability**
  - The application of models and traceability to the process presented complexity more gradually, separating concerns and providing guidance in performing the tasks. This allowed users to focus only on those parts impacted by the change request, increasing their efficiency and improving confidence in the implementation. In particular, the traceability aspect was felt to be extremely valuable as it focused attention on key points of value to the stakeholders. The traceability tool was integrated with the modeling tool, permitting navigation of traceability links and impact analysis of changing model elements.
  - The structure of the models (e.g. Requirements Model, Use Case Model, etc.) helped the team manage the complexity of the application and to structure their work. Additionally, if a team member had not worked on that aspect of the framework previously, the models allowed faster comprehension and thus earlier contribution to the team.

- **Development Process**
  - An MDD Process Framework such as the one used by Enabler supports rapid MDA adoption by the development team.
  - Many obvious tasks can be automated by transformations and by validations, allowing the developer to focus on higher value tasks, boosting morale, and supporting a focus on business knowledge.
  - Reusability in all phases of the methodology allows capitalization of acquired knowledge. This benefit is not only achieved for the software components and code, but also on domain concepts (i.e., retail concepts expressed in the meta-model).

- **Tools and Technologies**: Our tool chain helped the user to focus on his work instead of getting lost in tools, documentation and complex framework usage. It provided all the support necessary to work at the model level (OCL as model compiler, SVN plug in for merging models), and avoided context switching between tools since all tools were integrated into RSA on top of Eclipse.

References

Why Model Versioning Research is Needed!? 
An Experience Report*

Kerstin Altmanninger², Petra Brosch¹**, Gerti Kappel¹, Philip Langer², Martina Seidl¹, Konrad Wieland¹, and Manuel Wimmer¹

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

Abstract. The status of current model-driven engineering technologies has matured over the last years whereas the infrastructure supporting model management is still in its infancy. Infrastructural means include version control systems, which are successfully used for the management of textual artifacts like source code. Unfortunately, they are only limited suitable for models. Consequently, dedicated solutions emerge. These approaches are currently hard to compare, because no common quality measure has been established yet and no structured test cases are available. In this paper, we analyze the challenges coming along with merging different versions of one model and derive a first categorization of typical changes and the therefrom resulting conflicts. On this basis we create a set of test cases on which we apply state-of-the-art versioning systems and report our experiences.

Key words: model versioning, conflict categorization, tool evaluation

1 Introduction

With the increasing employment of model-driven engineering techniques for software development, the call for adequate infrastructural means supporting the effective management of software models grows ever louder. Tools successfully used for versioning textual artifacts like source code are only limited suitable for models, due to their line-oriented text comparison component.

The urgent need for a suitable infrastructure supporting effective model versioning has been widely recognized and first solutions start to emerge (cf. [3] for a survey and [1, 16, 18, 20] for model versioning approaches). In this young research area we are currently in such an early phase that even the research questions are not clearly stated and that goals and borders are not clearly defined, going seldom beyond the demand for precise conflict detection and supportive conflict

* This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-819584.
** Funding for this research was provided by the fFORTE WIT - Women in Technology Program of the Vienna University of Technology, and the Austrian Federal Ministry of Science and Research.
resolution. Furthermore, no established quality criteria are available. Hence, the evaluation and comparison of model versioning systems in a structured and comprehensible manner are hardly possible. Related work to this paper only focuses on particular phases of the versioning process as it is done in [10, 14]. Recently, Barett et al. [4] have performed an evaluation of the versioning capabilities of commercial modeling tools and provide an experience report. In this paper, we follow a similar approach with the difference that we start from a discussion of the phases passed through the merge process. This allows us to create a structured test set and categorize the open issues and problems of model versioning.

The paper is organized as follows. In Section 2 we propose a criteria catalogue consisting of test cases for the evaluation of model versioning systems. To establish a comprehensive test set, we analyze and categorize the phases run through during the merge of diverging working copies of one model. In Section 3 we apply state-of-the-art model versioning systems on the test cases in order to devise requirements and challenges for model versioning in Section 4 before we conclude the paper in Section 5.

2 Criteria Catalog for the Evaluation of Model Versioning Systems

In this paper, we aim at establishing test cases for the structured and repeatable evaluation of model version control systems. The test cases emerge from model versioning scenarios potentially leading to conflicts in order to test change detection and conflict detection facilities at various complexity levels. These scenarios are systematically categorized according to the phases of the versioning process and the different dimensions of the model merge problem. Due to space limitations we only discuss the categorization briefly and give some representative examples. For a complete description of the criteria catalog we kindly refer to our project homepage.

The most general layer of the categorization is set up according to the three phases of the versioning process, which are depicted on the right side of Figure 1. In the change detection phase the performed modifications on two working copies of one model are identified. The detected changes build the basis for the two subsequent phases, the conflict detection and inconsistency detection. In the second phase conflicts are detected by analyzing concurrent changes solely, whereas in the third phase consistency problems are revealed which would occur in the merged model incorporating all changes.

Change Detection. In this phase changes performed in parallel on the common base revision $V0$ resulting in the modified versions $V0'$ and $V0''$ are identified (cf. $\Delta$ in Figure 1). The change detection may be realized either in a state-based manner which considers only the final states of the modified versions or by a change-based approach where the modeling editor tracks the executed

---

1 http://www.modelversioning.org
operations directly [19]. However, the quality of the change detection directly correlates with the quality of the merged version.

We classify changes according to two orthogonal dimensions. The first dimension represents their dependency on an underlying modeling language. A change is *generic* if it may be applied to any model irrespectively of the modeling language. In contrast, a *specific* change depends on a certain metamodel. Consequently, a specific change is a specialization of a generic change. The second dimension considers the divisibility of an operation by distinguishing between *atomic* and *composite* changes.

From the resulting four combinations of these classifications, we omit *generic composite* because a composite operation makes only sense in the context of a specific modeling language. *Generic atomic changes* comprise the primitive atomic operations *add*, *delete*, and *update*. They may be performed on model elements (*add*, *delete*) and model properties (*update*) independently of the modeling language, i.e., the underlying metamodel. Generic atomic operations build the basis for more complex and language specific operations. *Specific atomic changes* are indivisible language-dependent operations like *rename* and *move*. The operation *rename* modifies a specific property which assigns—according to the underlying metamodel—a name to a model element. The operation *move* changes the containment of a model element which also requires knowledge on the underlying metamodel. Note that in certain environments *move* is realized as composite operation if allowed by the metamodel. The detection of *specific composite changes* like refactorings is challenging, but extremely important for the quality of the overall merge result [8, 12]. It enables a more compact representation of the difference report by folding atomic operations which belong to

![Diagram](image-url)

**Fig. 1.** Conflict categorization according to the phases of the versioning process.
The detection of the changes yields the basis for the conflict detection. Conflicts potentially occur whenever the modifications performed by two different modelers are overlapping, i.e., if the same model elements are involved. In some cases overlapping changes might not result in conflicts at all. Figure 2 (a) illustrates a test scenario in the domain of UML Class Diagrams where both modelers modify different properties, namely the visibility and the name, of the same attribute. It depends on the unit of comparison whether this situation ends up with a conflict or not. If the unit of comparison is the class or the attribute, then a conflict occurs. If the unit of comparison is more fine grained and the proper-
ties of the attribute elements are observed independently, no conflict should be reported.

**Conflicts.** Conflicts arise if parallel changes are overlapping within the same unit of comparison. Overlapping changes may either be equivalent or contradicting.

As indicated by its name, for equivalent overlapping changes two modelers perform the same modifications or different modifications with the same impact. Although such operations are overlapping, i.e., changing the same parts of the common base model, no conflict should be reported and only the modifications performed in one working copy should be included in the merged model. Figure 2 (b) illustrates a test scenario where both modelers add an equally named class inheriting from the same superclass.

Contradicting overlapping changes are concurrent modifications that do not commutate, i.e., their execution order affects the result [17]. Consequently, the operations cannot be merged and a conflict occurs. Such conflicts are caused by two concurrent update changes performed on the same property. Another kind of contradicting overlapping changes are concurrent delete and update operations involving the same model elements and properties as depicted in Figure 2 (c). Resolution of update/update conflicts as well as delete/update conflicts usually require manual interaction.

**Inconsistencies.** Finally, problems may occur regarding the consistency in the merged version of the working copies. According to the classification of Mens [19] we group such problems based on the level of language-specific representation.

If the merged model is not conforming to the metamodel or violates any other validation rules, a syntactic problem should be reported. Obviously, language-specific information, i.e., the metamodel and validation rules, has to be regarded to enable the detection of such problems [5]. Two examples of such a problem are shown in Figure 2 (d) and (e). In Figure 2 (d) both modelers add an inheritance relationship between the same classes but with different directions resulting in an inheritance cycle when merged. In the second example one modeler turns an association into a composition whereas the other increases the multiplicity at the same association end. Given these scenarios, a naive merge simply combines both modifications, resulting in an invalid model.

Although a merged model is valid, i.e., no syntactic problems are at hand, several issues may exist with respect to the static and/or operational semantics. Since the semantics and the correct interpretation of a model is difficult to express in a formal way, the detection of such problems is challenging. For instance, if two modelers express the same concept in linguistically different ways, a semantic-aware merge may prohibit an undesired merge result in which the same concept is contained twice. An example is given in Figure 2 (f) where one modeler shifts a method contained in two classes into their common superclass whereas the other modeler introduces a new subclass. Consequently, the new class inherits also the shifted method which might not be intended.
3 Evaluation of Model Versioning Systems

The categorization given in the previous section allows us to create test cases for the evaluation of conflict detection components in model versioning systems. In the following, we apply selected versioning systems on our test cases and discuss the results of the experiment.

Selected Versioning Systems. In previous work [3] we have conducted a survey on the state of the art of three-way merging approaches for model versioning systems. On basis of this survey, we have selected a set of versioning systems consisting of a text-based tool, a commercial tool, an open-source tool, and a tool developed in the context of a research project.

We have evaluated Subversion\(^2\) as representative of line-based versioning systems for text-based artifacts and three solutions dedicated to set model artifacts under version control. The IBM Rational Software Architect (RSA)\(^3\), a UML based model-driven development tool, provides a series of model management operations, including comparison and merge functions. The Eclipse plug-in EMF Compare [9] allows matching, comparing, and merging Ecore-based models. Finally, the CASE tool Unicase [13] provides a repository which is under version control. The Unicase client allows viewing and editing models in a textual, tabular, or graphical representation. The comparison algorithm uses the editing operations obtained from the Unicase client.

Experimental Setup. For this evaluation, we have focused on versioning UML Class Diagrams as this language is supported by all tools. The test cases cover all previously identified conflict categories. Due to space limitations we kindly refer to the AMOR project website\(^4\) for a detailed description.

To compare the quality of the conflict detection results of the selected tools, we reuse measures stemming from the field of information retrieval to compare the manually determined conflicts (the “relevant conflicts”) to the automatically found conflicts. The primary measures are precision and recall which are negatively correlated. Thus, we use a common combination of the primary measures, namely the F-measure. The measures are based on the notion of true-positives (tp), false-positives (fp), false-negatives (fn), and true-negatives (tn). In our evaluation we strictly rated the outcome in comparison to the expected result which we specified for each test case.

Results. Table 1 reports the results of our evaluation for representative test cases, summarized by the precision, recall, and F-measure values. The highlighted fields indicate that although other conflicts have been reported the expected result has not been accomplished. The precision values reflect the importance of operating on adequate representations of the graph-based models (cf.

\(^2\) http://subversion.tigris.org/
\(^3\) http://www-01.ibm.com/software/awdtools/architect/swarchitect/
\(^4\) http://www.modelversioning.org/
<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
<th>Subversion</th>
<th>RSA</th>
<th>EMF</th>
<th>Unicase</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Conflict</td>
<td>Add Different Class</td>
<td>fp</td>
<td>tn</td>
<td>tn</td>
<td>fp</td>
</tr>
<tr>
<td></td>
<td>Add Different Reference</td>
<td>fp</td>
<td>tn</td>
<td>tn</td>
<td>fp</td>
</tr>
<tr>
<td></td>
<td>Unit of Comparison</td>
<td>fp</td>
<td>tn</td>
<td>fp</td>
<td>tn</td>
</tr>
<tr>
<td></td>
<td>Rename vs. Move</td>
<td>fp</td>
<td>tn</td>
<td>fp</td>
<td>tn</td>
</tr>
<tr>
<td>Contradicting</td>
<td>Rename Class</td>
<td>tp</td>
<td>tp</td>
<td>tp</td>
<td>tp</td>
</tr>
<tr>
<td></td>
<td>Delete Rename Class</td>
<td>tp</td>
<td>tp</td>
<td>fn</td>
<td>tp</td>
</tr>
<tr>
<td></td>
<td>Delete Class vs. Add Method</td>
<td>tp</td>
<td>tp</td>
<td>tp</td>
<td>tp</td>
</tr>
<tr>
<td></td>
<td>Containment</td>
<td>fn</td>
<td>fn</td>
<td>tp</td>
<td>fn</td>
</tr>
<tr>
<td>Equivalent</td>
<td>Add Same Class</td>
<td>fn</td>
<td>fn</td>
<td>tp</td>
<td>fn</td>
</tr>
<tr>
<td></td>
<td>Contradiction in Hierarchy</td>
<td>fn</td>
<td>fn</td>
<td>fn</td>
<td>fn</td>
</tr>
<tr>
<td>Syntax</td>
<td>Dangling Reference</td>
<td>fn</td>
<td>tp</td>
<td>tp</td>
<td>tp</td>
</tr>
<tr>
<td></td>
<td>Inheritance Cycle</td>
<td>fn</td>
<td>tp</td>
<td>fn</td>
<td>fn</td>
</tr>
<tr>
<td></td>
<td>Contradiction in Reference</td>
<td>fn</td>
<td>tp</td>
<td>fn</td>
<td>fn</td>
</tr>
<tr>
<td>Semantic</td>
<td>Inheritance of Methods</td>
<td>fn</td>
<td>fn</td>
<td>fn</td>
<td>fn</td>
</tr>
<tr>
<td></td>
<td>Semantics in Associations</td>
<td>fn</td>
<td>fn</td>
<td>fn</td>
<td>fn</td>
</tr>
<tr>
<td>Measures</td>
<td>Precision</td>
<td>0.43</td>
<td>1</td>
<td>0.71</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>Recall</td>
<td>0.27</td>
<td>0.55</td>
<td>0.45</td>
<td>0.36</td>
</tr>
<tr>
<td></td>
<td>F-Measure</td>
<td>0.34</td>
<td>0.71</td>
<td>0.55</td>
<td>0.47</td>
</tr>
</tbody>
</table>

Subversion vs. RSA. In contrast to the high precision values, the recall values are less promising, indicating that certain conflicts are not automatically detectable at all influencing the F-measures values. In the following we report our experiences with the different tools.

Subversion. We used Subversion to put the serialized models under version control. This has been directly done in the file system independently of the modeling editor. For the conflict resolution we have used the default text-based tool which marks and shows the location of the conflicting changes in the files. First, we have tried to version XMI-serializations of models created with the Enterprise Architect (EA). Since the EA includes much additional information like on graphical representation and time stamps in the XMI-files, many conflicts have been reported which are not related to our artificially provoked conflicts. Hence, the application of Subversion has been impractical for this kind of serialization.

Second, we have tried to version XMI-files created with the Eclipse UML 2.1 plug-in\(^5\) which serializes the models in two files: one containing the actual model information and one containing information on the graphical representation. Due to the very generic comparison algorithm, the conflict report confines itself on indicating modifications occurring at the same location in the text file. For example, in the “Add Different Class” test case, a conflict is reported, as the new classes are inserted at the same position. Also in the “Add Same Class” test case a conflict is reported, as the inserted classes have different IDs. If not the same lines are affected, the merge is performed without any further inquiry. This may result in invalid models which, e.g., contain compositions with a multiplicity

---

\(^5\) [http://www.eclipse.org/uml2](http://www.eclipse.org/uml2)
higher than one or dangling references. Subversion naturally provides no specific validation mechanisms.

Overall, using Subversion showed up as being as expected, i.e., even for small examples the conflict resolution is challenging and error-prone due to missing tool support.

*Rational Software Architect (RSA).* The “Compare with each other” command of the RSA enables the comparison between three UML models where one of them has to be declared as ancestor. The differences of the models V0’ and V0” are presented in separate windows and a preview of the merged version is provided. The RSA also reports changes concerning the graphical representation of the model.

The RSA supports the merging process in a semi-automatic manner because manual interaction is always required even if no conflict is reported. When conflicts occur, the user must decide which change—either from the left or from the right model—to perform first. Obviously, the resolution order influences the result. Furthermore, the RSA also provides the two language-specific operations *rename* and *move*. Thus, contradicting changes caused by these operations are handled without any problems. After resolving all conflicts the user has to overwrite either the left or the right working copy. The RSA also provides a validation mechanism to detect consistency problems concerning the syntax of the merged model. Unfortunately, the user may only validate the new merged version of the model after overwriting one of the working copies.

Overall, equivalent changes like “Add Same Class” are not identified as conflicts; thus, both classes exist in the merged version. The same is true for consistency problems concerning the semantic of models. The RSA detects contradicting changes in a fine-grained manner, but only if dedicated, language-specific rules exist.

*EMF Compare.* For using the three-way merge of EMF Compare three different files of EMF-based models need to be selected from the Eclipse workspace. With the command “Compare with each other” and the according selection of the origin model the differences and conflicts between the origin model and two modified versions are visualized in a tree-based manner. Additionally to the tree-based presentation, the differences and conflicts are visualized between the edited model versions (V0’ and V0”) with colors (blue for changes and yellow for conflicts). To merge the parallel evolved models changes can be “copied” from the left model artifact (V’) to the right one (V”) and vice versa. A special option exists to just copy the non-conflicting changes.

As stated in Table 1, about half of the test cases result in accurate conflict detection reports. Since EMF Compare does not incorporate the metamodel of a modeling language nor offers language-specific extensions, it is not able to consider language-specific constructs (e.g., inheritance, constraints, refactorings). Hence, especially for the test cases belonging to the categories syntax and semantics EMF Compare determines a series of false-negative results. Moreover, since the unit of consistency for comparison cannot be adapted according to
specific languages, false-positive results may occur like in our test cases in the “Unit of Consistency” and “Rename vs. Move” examples.

Summing up, EMF Compare offers an adequate conflict detection report for versioning any kind of EMF-based model artifacts but has naturally deficiencies in detecting language-specific conflicts.

*Unicase.* The CASE tool Unicase incorporates versioning support using a central repository. When checking in a model based on an outdated version, the conflict detection process is started. Although conflicts are not reported explicitly, the merge of two concurrently modified versions is intuitive and does not allow to apply conflicting changes. When merging, all changes of both sides are listed in two columns. The user may now choose which changes she likes to apply to the merged version. If the user selects a change, all conflicting changes are automatically unselected and highlighted in the opposite side. Two versions are never automatically merged without user interaction even if there is no conflict.

Changes are directly tracked while the user modifies a model. Consequently, the detection of atomic changes works precisely. However, composite changes like refactorings are not detected and as a result, they are not regarded in the conflict detection process. In general, Unicase detects delete/update conflicts as well as update/update conflicts using a fine-grained granularity (element property). Any concurrent modification of the same property may not be merged. Although this is straightforward in most of the cases, e.g., concurrent renaming of a class, it often leads to an undesired behavior. For instance, Unicase does not allow to apply all changes of both sides if two classes have been concurrently added in the same package because the containment property of the package has been updated on both sides. Unfortunately, this has often resulted in *false-positives* in our evaluation even though Unicase generally provides an accurate conflict detection in the first two categories. Language-specific conflicts like syntactic problems have never been reported. Even a manual validation after the merge has not reported validation problems which shows that the metamodel used by Unicase does not seem to offer detailed validation rules.

To sum up, Unicase offers an adequate conflict detection on a metamodel level. Generic atomic changes and their potential contradictions are detected correctly. However, specific composite changes and conflicts which require modeling language specific knowledge remain unrevealed.

## 4 Lessons Learned

As the evaluation results show, unreliable tool support for model versioning forms an insurmountable obstacle for the professional application of MDE. In the following, found issues and lessons learned from the evaluation are summarized and research questions to be answered for tackling those problems are stated.

**Benchmark Availability.** To the best of our knowledge, there is currently no common benchmark for model versioning systems available in the related work. Hence, neither detailed requirements nor the expected run-time behavior
Unreliable Conflict Detection. A major deficiency is the unreliable conflict detection. False-positives as well as false-negatives occur regularly already in our small test cases. False-positives are often reported due to updated containment properties, since conflict detection mostly works on metamodel level. Furthermore, semantic inconsistencies are hardly detected as such problems are usually ignored. Research for answering the question “What is the expected result?” is needed for improving the overall check-in process.

Confusing Difference Report. The representation of changes in concurrently edited models differs from tool to tool. Since concurrent changes are not visualized by using the model’s concrete syntax, but by presenting a list or a tree structure of atomic changes, those metamodel based difference reports are not intuitive and rather confusing for modelers. For providing the modeler a better understanding of what happened, two improvements are necessary. First, differences and conflicts should be presented on the modeler’s point of view, i.e., on model level. Second, related atomic changes should be grouped as one composite change.

Aggregating atomic changes to composite changes in generic modeling environments is no trivial task because of the numerous combination possibilities. To tackle this problem, Brosch et al. [8] present an operation recording approach allowing the user to define language-specific composite operations by modeling small examples. Weber et al. [21] define composite change patterns for process-aware information systems in order to reduce the complexity of process changes while raising the level of expressiveness. Küster et al. [15] compute critical pairs of dependencies and conflicts for compound change operations in process models.

Single Diagram Support. Even if some specific diagram types (e.g., UML Class Diagram) are well supported by tools, extensive application of optimistic model versioning in MDE projects does not prove satisfactory. Current development strategies include employment of specific modeling languages for specific problem domains. Like Subversion [11] for the versioning of arbitrary text files, model versioning systems must provide support for arbitrary modeling languages. For making use of the rich semantics of the model’s graph-based nature, language-specific adaptation of generic model versioning systems is a key research field as it is done by Altmanninger et al. [2]. Furthermore, conflicts involving multiple types of diagrams have to be considered.

Unreliable Conflict Resolution. We have learned from the evaluation that due to the open challenges in conflict detection any support for automatic conflict resolution is only a vision and current tools only support manual decision of approving changes from left or from right. This manual approach is not only cumbersome but error-prone as well. Unfortunately, checks for ensuring correct syntax and semantics of the merged version were seldom performed in the testbed. Thus, the consistency of the merged version is not guaranteed.
5 Conclusion

In this paper, we presented a first categorization of conflicts occurring during the check-in process in model versioning systems. On this basis we inferred numerous test cases which allowed us to conduct experiments with state-of-the-art tools. Our tests enabled us to compare the tools in a structured and fair manner. The results were far from satisfying, but promising. For the moment, we focused on the UML Class Diagram, but in future work we will extend the test set with test cases containing other kinds of diagrams like UML State Charts or Activity Diagrams. The long-term objective is to establish a comprehensive, expressive benchmark for the evaluation of model versioning systems which covers a multitude of different scenarios.

Overall, many interesting research questions are open in the young research area of model versioning, demanding for a common terminology and an exact formulation of the research goals. We are aware that the model versioning test cases presented in this paper are only a drop in the ocean. But in fact, having test cases is a corner stone for structured research and we are looking forward to investigate the open issues together with the model versioning research community.

References

Limitations of Automating Model Migration in Response to Metamodel Adaptation

Markus Herrmannsdoerfer and Daniel Ratiu

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

Abstract. In consequence of changing requirements and technological progress, modeling languages are subject to change. When their metamodels are adapted to reflect those changes, existing models might become invalid. Manually migrating the models to the adapted metamodel is tedious. To substantially reduce effort, a number of approaches have been proposed to fully automate model migration. However, modeling language evolution occasionally leads to metamodel changes for which the migration of models inherently cannot be fully automated. In these cases, the migration of models requires information which is not available in the model. If such changes are ignored or circumvented, they may lead to language erosion. In this paper, we formally characterize metamodel adaptations in terms of the effort needed for model migration. We focus on the problem of metamodel changes that prevent the automatic migration of models. We outline different possibilities to systematically cope with these kinds of metamodel changes.

1 Introduction

Even though often neglected, a modeling language is subject to change like any other software artifact [1]. This holds for both general-purpose and domain-specific modeling languages. For instance, UML [2] – a general purpose modeling language – already has a rich evolution history, although it is relatively young. Domain-specific modeling languages – like e.g. AUTOSAR [3] for the specification of automotive software architectures – are even more prone to change, as they have to be adapted, whenever their domain changes due to technological progress or evolving requirements.

A modeling language is evolved by adapting its metamodel to the evolved requirements. Due to metamodel adaptation, existing models may no longer conform to the adapted metamodel. These models have to be migrated so that they can be used with the new version of the modeling language. Throughout the paper, the combination of metamodel adaptation and reconciling model migration is referred to as coupled evolution. Manually migrating existing models to the adapted metamodel is tedious and error-prone. Adequate tool support is thus required to reduce the effort for model migration.
To determine requirements for adequate tool support, we reverse engineered the coupled evolution of a number of real-world modeling languages [4, 5]. More specifically, we classified the performed metamodel changes according to the automatability of the associated model migration. Metamodel-only changes do not require the migration of models, whereas coupled changes do. Metamodel-independent coupled changes do not depend on a specific metamodel. Metamodel-specific coupled changes are so specific to a certain metamodel that the migration cannot be reused across metamodels. Model-specific coupled changes require information from the modeler during migration, and thus the migration cannot be specified in a model-independent way. As no changes were classified as model-specific, we decided to ignore them for the first version of our tool support called COPE. A great threat to the study’s validity is that the developers might have feared and thus avoided model-specific coupled changes.

With adequate tool support at our hands, we started using COPE to forward engineer the coupled evolution of a number of modeling languages. From our experience with adapting several metamodels, we learned that sometimes changes requested by the users of the modeling language require a metamodel adaptation that prevents the automation of the model migration. Using our terminology presented above, we would say that these cases require a model-specific coupled evolution. Implementing such a change would invalidate the existing models, which have to be migrated manually. However, manually migrating a potentially unknown number of models imposes a heavy burden on the language users. Consequently, the language developers are tempted to avoid model-specific coupled evolution by adapting the metamodel in a way that the model migration does not require information from the users. However, not being able to implement such changes limits modeling language evolution, and threatens the simplicity and quality of the metamodel. Unfortunately, existing approaches are not able to cope with model-specific coupled changes.

In this paper, we provide a formal framework to characterize metamodel adaptations in terms of the efforts needed for model migration. We focus on the problem of metamodel changes that prevent the automatic migration of models. We outline different possibilities to systematically cope with these kinds of metamodel changes.

Outline. In Sec. 2, we provide a number of examples to distinguish model-specific coupled changes from other kinds of changes. In Sec. 3, we present the formal framework to generally characterize model-specific coupled evolution. In Sec. 4, we outline a number of possible solutions to cope with model-specific coupled evolution In Sec. 5, we analyze related work with respect to their support for model-specific coupled evolution, before we conclude in Sec. 6.

2 Motivating Example

We chose a simple modeling language to specify automata as a running example. To show different levels of automating a model migration, we present a number
of adaptations to the language’s metamodel. Fig. 2 depicts the different versions of the metamodel as UML class diagrams.

Fig. 1(a) depicts the original version of the metamodel. An Automaton defines a number of states each of which is identified by a name. A State has a number of outgoing transitions each of which is activated by a trigger. When a Transition is activated, the control transitions to a target state. When the target state is entered, a sequence of actions is performed as effect. Strings are used for state names and to denote triggers and effects.

As is depicted in Fig. 1(b), we introduce hierarchical states to our modeling language to be able to structure complex automata. In the metamodel, we thus make Automaton a subclass of State. Now, a State can also be an Automaton which can again be decomposed into a number of states, and so on. Existing models are not affected by this metamodel adaptation, as they are still valid and have the same meaning. As a consequence, they do no have to be migrated, and the adaptation basically is a conservative extension.

Fig. 1. Metamodel Adaptation

---

1 For better overview, metamodel adaptations are highlighted by dashed boxes in the figure.
As is depicted in Fig. 1(c), we change the modeling language from a Moore to a Mealy automaton to ease the specification of effects. In Moore machines, the effect of the automaton only depends on the current state. In contrast, the effect of the automaton depends also on the trigger in Mealy machines. In the metamodel, we thus move the attribute `effect` from `State` to `Transition`. In response, models no longer conform to the adapted metamodel, as an effect is no longer allowed for a state. To migrate these models, the `effect` of each state has to be moved to all its incoming transitions. Note that this model migration can be fully automated by the well-known algorithm to transform Moore automata to Mealy automata.

As is depicted in Fig.1(d), we introduce initial states to refine the modeling language. In the metamodel, the reference `initial` is introduced to denote the initial state within an automaton. As this reference is mandatory, the model needs to be migrated to add the missing information. However, the initial states cannot be unambiguously inferred from the information already available in the model, and default values cannot be provided either. As a consequence, this model migration cannot be automated, as information is required from the developer of the model during migration.

One solution would be to avoid the model-specific change. For example, we could make the new reference `initial` optional instead of mandatory in order to not invalidate existing models. However, our metamodel would not be as robust as we want it to be, and our tools would have to cope with missing initial states in an ad-hoc manner (each tool can interpret the initial states differently) and this would lead to ambiguities. As a result, this reduces the overall quality and simplicity of the modeling language and makes models fragile. To prevent such language erosion, we need to be able to support model-specific coupled evolution.

3 Characterizing Model-specific Migration

Language definition. Before we can characterize evolution, we have to formally define our understanding of a modeling language. A modeling language is completely specified through its abstract syntax and semantics [6].

Definition 1 (Modeling language). A modeling language \( \mathcal{L} = (\mathcal{M}, \mathcal{S}) \) is a tuple of abstract syntax \( \mathcal{M} \) and semantics \( \mathcal{S} \). The abstract syntax \( \mathcal{M} = \{m_1, m_2, \ldots\} \) defines the (possibly infinite) set of all models that are syntactically correct. The semantics \( \mathcal{S} : \mathcal{M} \rightarrow SD \) is defined by a mapping from the models \( \mathcal{M} \) to a semantic domain \( SD \).

A model is syntactically valid if it is built from the constructs and fulfills the constraints defined by the metamodel. For the sake of simplicity, we omit the formalization of the relationship between model and metamodel here. Two models \( m_1, m_2 \in \mathcal{M} \) are syntactically equivalent if and only if \( m_1 = m_2 \). In practice, usually only a small subset \( \mathcal{M}_{built} \subset \mathcal{M} \) of the possible models are actually built. Two models \( m_1, m_2 \in \mathcal{M} \) are semantically equivalent, i.e. \( \mathcal{S}(m_1) \equiv \mathcal{S}(m_2) \), if the interpretation returns equivalent objects from the semantic domain.
Language evolution. A language evolution is usually reflected by the adaptation of the metamodel and/or in the adaptation of the semantic mappings. In the following, we talk about two language versions $L_1 = (M_1, S_1)$ and $L_2 = (M_2, S_2)$, where $L_1$ evolved to $L_2$. If the language evolution is an extension, we do not need to migrate existing models.

Definition 2 (Conservative extension). $L_2$ is a conservative extension of $L_1$ if and only if $M_1 \subseteq M_2$ and $S_1(m) \equiv S_2(m)$ for all $m \in M_1$.

A conservative extension thus does not syntactically invalidate existing models, and preserves their meaning. An extension can occur when new constructs are added to the metamodel, when existing constructs are extended, or when constraints are removed or weakened. For instance, in Fig. 1(b), we extended the modeling language with hierarchical states. Existing flat automata are still valid with respect to the adapted metamodel, and have the same meaning.

Definition 3 (Restriction). $L_2$ is a restriction of $L_1$ if and only if $M_2 \subseteq M_1$.

A restriction syntactically invalidates a number of models, namely $M_1 \setminus M_2$, which have to be migrated. A restriction can occur when constructs are removed from the metamodel, when existing constructs are restricted, or when constraints are added or strengthened. For instance, evolving from Fig. 1(b) back to Fig. 1(a), is a restriction, as automata with hierarchy are no longer valid and thus have to be flattened.

In general, a new version of the language exhibits both restrictions and extensions which we call a variation.

Definition 4 (Variation). $L_2$ is a variation of $L_1$ if $M_2 \setminus M_1 \neq \emptyset$ and $M_1 \setminus M_2 \neq \emptyset$.

A variation invalidates outdated models, namely $M_o = M_1 \setminus M_2$, which have to be migrated. In practice, only those outdated models need to be migrated that are actually built, i.e. $M_{built} \cap M_o$. For instance, in Fig. 1(e), we change the language from a Moore to Mealy automata. All models that define effects for their states are no longer valid, but we now allow new models that define effects for their transitions. We have to find a mapping that migrates the outdated models to the new language version, and at the same time preserves the semantics of the models.

Model migration. As we have seen before, models may have to be migrated to preserve them in response to language evolution. A model migration is defined by a migration function which maps the models from the old language $L_1$ to models of the new language $L_2$.

Definition 5 (Migration function). A migration function is a function $\mu : M_1 \rightarrow M_2$ which maps each model $m \in M_1$ to a model $\mu(m) \in M_2$ such that $S_1(m) \equiv S_2(\mu(m))$. 
Note that we explicitly require that the migration function preserves the semantics of all models. Otherwise, information is lost during migration which needs to be avoided. For instance, the transition from a Moore to a Mealy automaton (see Fig. 1(c)) requires a migration function that maps all models with effects in states to models with effects in transitions. These models have to be mapped in a way that the effects are moved for each state to all incoming transitions.

However, there are cases in which the mapping cannot be performed without information which is not available in the model.

**Definition 6 (Model-specific migration).** A migration \( \mu \) is model-specific if and only if a model \( m_1 \in M_1 \) exists for which \( |E(m_1)| > 1 \), where \( E(m_1) = \{ m_2 \in M_2 \mid S_1(m_1) \equiv S_2(m_2) \} \).

A migration has to be model-specific if several models in \( M_2 \) exist which are semantically equivalent to a model in \( M_1 \). In this case, additional information is necessary during migration to choose one of these semantically equivalent models. The set for which the migration has to be model-specific is \( M_{ms} = \{ m_1 \in M_1 \mid |E(m_1)| > 1 \} \). Again, model-specific migration is only required for models from \( M_{built} \cap M_{ms} \).

For instance, in Fig. 1(d), we introduced mandatory initial states which leads to a model-specific migration. A number of models with different choices for initial states are semantically equivalent to each model without initial states. Model-specific information is required to choose the one that is intended by the developer of the model.

A migration is thus model-independent if \( |E(m_1)| = 1 \) for all \( m_1 \in M_1 \). For instance, for the transition from a Moore to a Mealy automaton (see Fig. 1(c)), there is exactly one model in \( M_2 \) that is semantically equivalent to a model in \( M_1 \). Consequently, no additional information is required during migration, and we can thus specify an algorithm that automatically performs the migration.

### 4 Coping with Model-specific Migration

We outline a number of possible solutions to cope with model-specific coupled evolution. The solutions take advantage of particular situations which are however encountered very often in practice.

**Effort analysis.** In practice, only a small subset of all possible models \( M \) are actually built. Only the existing models need to be migrated. In many practical situations, (e.g. for highly domain-specific languages) the language developers and users are quite close to each other (e.g. in the same company). In these situations, it is often the case that the entire set of the existing models is known to the language developers. For instance, whenever the language is used only in-house all models are contained in a central repository.

In case when all models are known, language developers can make informed language improvements also with respect to the effort needed for model migration. Based on the existing models, they can assess the manual migration effort...
required after metamodel adaptation. For instance, the effort needed to introduce initial states is proportional to the number of automata in the existing models. They can decide whether an improvement in the language is worth making given the amount of manual work necessary to migrate the existent models. However, in a lot of cases, language developers and users are decoupled which makes this approach infeasible.

*Interactive migration.* One possibility to support model-specific coupled evolution is to provide user interaction during migration. The migration algorithm automatically migrates the model as far as possible, and whenever it needs supplementary information, it asks the language user to provide the missing information. It can also suggest a number of alternatives from which the language user has to choose. We have extended the language provided by our approach COPE with a primitive to trigger such an interaction during migration.

Listing 1.1 shows the use of this primitive in a coupled operation that introduces initial states in the language. The metamodel adaptation creates the new reference `initial` which is single-valued and mandatory. During model migration, the developer of the model has to choose an initial state for each automaton. The primitive `choose` takes three parameters as input, namely the context element, the values to choose from and a message, and returns the chosen value.

Listing 1.1. Interactive Coupled Operation

```java
// metamodel adaptation
newReference(Automaton, "initial", State, 1, 1, !CONTAINMENT)

// model migration
for (a in Automaton.allInstances) {
    a.initial = choose(a, a.state, "Choose initial state")
}
```

Figure 2 shows the dialog that is opened during model migration to let the language user make a choice. The dialog shows the current state of the model selecting the context element, the list of values to choose from, as well as the message. The developer of the model is required to choose a value from the list to determine the initial state within an automaton.

*Implicit information.* Many times, the language users employ different conventions (e.g. naming conventions) to capture more information than is made explicit through the metamodel. In these cases, the language user can incorporate implicit information to help automate the migration. For example, language users might have already named all initial states with a prefix "I", before the initial states were explicitly introduced in the metamodel. They can then refine the migration in a way that for each automaton, it automatically chooses the
marked sub state. We thus plan to introduce a means to allow the language user to upfront establish certain choices introduced by the language developer.

5 Related Work

Current approaches to reduce the effort for model migration focus on fully automating the model migration. They thus do not address model-specific coupled evolution which can inherently not be fully automated. We believe that the current language evolution approaches do not support model-specific migration, because the problem is not completely understood yet. With this paper, we thus aimed to characterize the problem, and outlined possible solutions. Related work on model migration can be subdivided into two kinds of approaches: difference-based and operation-based.

**Difference-based approaches** allow language developers to synthesize a model migration based on the difference between two metamodel versions. Sprinkle et al. introduce a visual graph-transformation-based language which requires to specify the model migration only for the metamodel difference [7]. However, this language does not provide constructs to cater for model-specific migrations. The following approaches further automate model migration by automatically deriving it from the difference between two metamodel versions. Gruschko et al. classify primitive metamodel changes into non-breaking, breaking resolvable and unresolvable changes [8, 9]. Based on this classification, they propose to automatically derive a migration for non-breaking and resolvable changes. Cichetti et al. go even one step further and try to detect composite changes like e.g. extract class in the difference between metamodel versions [10, 11]. Garces et al. refine this approach, present a prototype, and demonstrate its applicability in
two case studies [12]. However, the derivation approaches are not able to detect model-specific migrations in the metamodel difference.

**Operation-based approaches** allow language developers to incrementally transform the metamodel by means of coupled operations which also encapsulate the corresponding model migration. Although they are not as automated as difference-based approaches, they allow to capture the intended model migration already when adapting the metamodel. Hößler and Soden present a number of high-level coupled operations which automate metamodel adaptation as well as model migration [13]. Wachsmuth adopts ideas from grammar adaptation and proposes a classification of metamodel changes based on instance preservation properties [14]. Based on these preservation properties, the author defines a library of high-level coupled operations. In [5, 15] we generalize these approaches by a language which allows to specify arbitrary new high-level coupled operations. Moreover, this language can be used to attach a custom migration to a metamodel adaptation, which cannot be covered by high-level coupled operations. We demonstrate the usefulness of this approach by presenting the coupled evolution of two real-life metamodels. However, all of these approaches do not provide coupled operations which support model-specific migrations. In this paper, we thus extended our language for expressing coupled operations with a choice construct that allows the user to guide the migration.

6 Conclusion

Modeling languages evolve over time and thereby their metamodels need to be adapted. To reduce the effort for language evolution, the resulting migration of models needs to be automated. However, some metamodel changes require information during migration which is not available in the model. Consequently, these metamodel changes inherently prevent the automatic migration of models. In this paper, we presented an example of such model-specific changes and formally characterized them. Moreover, we presented a number of techniques to cope with model-specific migrations. With these techniques, language developers can make informed decisions about the effort needed for manual migration, and partially automate the manual migration by means of adequate tool support. As a consequence, language erosion can be prevented which can result from avoiding model-specific changes. It remains to be investigated how much a language suffers from systematically avoiding them. Moreover, we are interested in how often model-based changes are required in practice, when not avoiding them.

References

4. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Automatability of Coupled Evolu-
5. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - Automating Coupled Evolu-
7. Sprinkle, J., Karsai, G.: A domain-specific visual language for domain model evolu-
fication Scheme for Semi-Automatic Meta-Model Evolution. In: 1st Work-
shop MDD, SOA und IT-Management (MSI), GI, GiTO-Verlag (2007) 35–46
11. Cicchetti, A., Ruscio, D.D., Pierantonio, A.: Managing dependent changes in cou-
12. García, K., Jouault, F., Cointe, P., Bézivin, J.: Managing model adaptation by pre-
13. Höfler, J., Soden, M., Eichler, H.: Coevolution of Models, Metamodels and Trans-
Model Patches in Model-Driven Engineering

Antonio Cicchetti¹, Davide Di Ruscio², and Alfonso Pierantonio²

¹ School of Innovation, Design and Engineering
Mälardalen University,
SE-721 23, Västerås, Sweden
antonio.cicchetti@mdh.se

² Dipartimento di Informatica
Università degli Studi dell’Aquila
Via Vetoio, Coppito I-67010, L’Aquila, Italy
{diruscio|alfonso}@di.univaq.it

Abstract. Increasingly, recording the various kinds of design-level structural evolution that a system undergoes throughout its entire life-cycle is gaining relevance in software modeling and development. In this respect, an interesting and useful operation between subsequent system versions is model difference consisting in calculation, representation, and visualization. This work shows how to generalize the application of differences, represented as first-class artefacts, in order to abstract from persistent identifiers and enable more flexibility. Then, modifications can be applied as model patches to arbitrary models according to weaving specifications.

1 Introduction

Model Driven Engineering (MDE) [1] emerged to increase productivity and reduce time-to-market by enabling development using concepts closer to the problem domain, rather than those offered by programming languages. Moreover, it is aimed at making the software assets more resilient to changes caused by the emerging technologies and makes the role of modeling and models in the current software development much more important. Similarly to what happened for source code, versioning techniques are increasingly needed for supporting the evolution of model-based artefacts. In this respect, the detection of differences between models is essential to model development and management practices [2, 3].

The problem of model differences is intrinsically complex and requires specialized algorithms and notations [4, 5]. Recently, in [6, 7] two similar techniques have been introduced to represent differences as models, hereafter called difference models; interestingly, in these proposals difference models are declarative and enable the reconstruction of the final model by means of automated transformations which are inherently defined in the approaches.

In this paper, we enhance the work in [6] in order to support the application of model differences as model patches, that is to apply the modifications described in a

* Partially supported by the European Community’s 7th Framework Programme (FP7/2007-2013), grant agreement n° 214898.
difference model to an arbitrary model. For instance, in Fig. 1 the difference model $\Delta$ detected between $M_1$ and $M_2$ is applied to reproduce the changes over $M'_1$ to obtain $M'_2$ with $M'_1$ obtained from $M_1$ by means of some manual intervention. This is usually difficult because of persistent identifiers which refer to specific model elements limiting the application of the differences to the original models only, and sometime even to the same modeling environment [8]. The proposed approach considers differences as model patches analogously to Larry Wall’s work on Unix patch 3. Specific techniques based on the concept of weaving models [2] are used to abstract from persistent identifiers and to explicitly identify those model elements over which modifications must be applied.

The paper is structured as follows: Sect. 2 outlines the problem of model difference management, whereas Sect. 3 proposes the approach to apply model differences as patches. Section 4 compares this paper with others. Finally, conclusions and perspective works are drawn.

2 Background

The overall structure of the technique in [6] is depicted in Fig. 2: given two base models $M_1$ and $M_2$ which conform to an arbitrary base metamodel $MM$, their difference conforms to a difference metamodel $MMD$ derived from the former by means of an automated transformation $MM2MMD$. The approach does not impose any restriction over the metamodel $MM$, i.e. it is metamodel-independent and can be applied to any arbitrary modeling language. In particular, the metamodel extension implemented in the $MM2MMD$ transformation consists of adding new constructs able to represent the possible modifications that can occur on models and which can be grouped as follows:

- **additions**: new elements are added in the initial model like the abstract class Figure in the sample model $M_2$ depicted in Fig. 3.b not present in the initial version of the specification (see the model $M_1$ in Fig. 3.a);
- **deletions**: some of the existing elements are deleted as a whole like in $M_1$, where the class Position is not present in $M_2$ anymore;
- **changes**: a new version of the model being considered can consist of updates of already existing elements. For instance, the structural features (associations and operations) of the Circles and Polygons classes in Fig. 3.a have been modified giving place to the new version $M_2$.

---

As mentioned, in order to represent the differences between $M_2$ and $M_1$, their base metamodel has to be extended by applying the MM2MM transformation. Essentially, for each metaclass $MC$ of the source metamodel, the additional metaclasses AddedMC, DeletedMC, and ChangedMC are generated. For instance, the metaclass Class of the UML metamodel induces the generation of the metaclasses AddedClass, DeletedClass, and ChangedClass. In the same way, the metaclass Operation entails the addition of the new specializations that are AddedOperation, DeletedOperation, and ChangedOperation.

The generated difference metamodel is able to represent all the differences amongst models which conform to the base metamodel. For instance, the difference model in Fig. 4 (that will be called $\Delta_1$ hereafter) represents the differences between the models $M_1$ and $M_2$ in Fig. 3. The differences depicted in such a model can be summarized as follows:

1. the addition of the new class Figure in $M_2$ is represented by means of the AddedClass instance $c_1$ which is associated with the instance $o_1$ of the metaclass AddedOperation;
2. the addition of the new class Group is represented through the instance $c_2$ of the AddedClass metaclass. The method draw is represented like that of the new Figure class. Moreover, the composition between the Group and Figure classes is represented through $assoc_1$ which is an instance of the AddedAssociation metaclass;
3. in the new version of the model, the Circles and Polygons classes are specializations of the class Figure. These modifications are represented by means of the instances $c_3$ and $c_5$, respectively, of the ChangedClass metaclass. Each of them refer to the corresponding updatedElement which represents the new version of the element. For instance, the instance $c_4$ of the metaclass Class has the new parent association with the class Figure and does not have the attribute pos which has been deleted in the new version of the class Circles;
4. the deletion of the class Position has been represented by means of the instance $c_7$ of the DeletedClass metaclass. Moreover, the attributes $x$ and $y$ of the same
Fig. 3. Two versions of the same model

The proposed difference representation meets a number of requirements which are given in [6]; in particular, in this work we exploit the self-containedness property to extend the application of manipulations to arbitrary models conforming to the base metamodel $MM$. 

2.1 Difference Application

According to Fig. 2 the model transformation $MMD_{MM2MM}$ is generated to apply the differences stored in a difference model $\Delta$ to an initial model $M_1$ in order to obtain a final $M_2$. $MMD_{MM2MM}$ is automatically derived by means of the higher-order transformation $MMD2ATL$ in Fig. 2, i.e. a transformation which produces another transformation [2]. In particular, $MMD_{MM2MM}$ consists of rules which apply over a model $M_1$ the additions, deletions and changes specified in $\Delta$.

The application of the transformation $MMD_{MM2MM}$ to the models $M_1$ and $\Delta_{1,2}$ reconstructs the final model $M_2$ in Fig. 3.b. Nevertheless, it may be very convenient to be able to apply a difference to any model conforming to the base metamodel. In other words, we want to use difference models as model patches analogously to Larry Wall’s Unix patch. To the best of our knowledge, in current model-based approaches (e.g. [9–11, 7]) difference application is either not supported at all or limited to those models which have been used for the difference calculation. In this respect, the approach illustrated so far in Fig. 2 is no exception. In fact, the generated $MMD_{MM2MM}$ transformation is exact, i.e. does not comprise any adjustability of its application. This problem, that will be tackled in the next section, is mainly due to the persistent identifiers, that is identifiers which are assigned to model elements by the modeling tools. They compromise interoperability and lock the models within a specific platform since identifiers are not universally computable. For instance, $\Delta_{1,2}$ embeds the references to the model elements of $M_1$ and $M_2$ involved in the calculation of the differences. Hence, the application of $\Delta_{1,2}$ to a model different than $M_1$ (like the scenario shown in Fig. 1), probably will result either in an idempotent application or in an erroneous model since the involved persistent identifiers could be no longer valid.

The next section generalizes the application of model difference to arbitrary models by introducing weaving models [2], which can be considered morphisms between models, to support the realization of model differences as model patches. This will enable,
for instance, the application of $\Delta_{1,2}$ as is also to a model $M'_1$ obtained from $M_1$, thus requiring that the refactoring originally operated from $M_1$ to $M_2$ has to involve also new elements while leaving all the other modifications still valid.

3 Modeling difference application

In this section, the metamodel independent approach outlined above is extended in order to model the application of differences to arbitrary models. For instance, starting from the manipulation of $M_1$ as shown in Fig. 5.a, it shall be possible to apply the differences stored in $\Delta_{1,2}$ in order to obtain a $M'_2$ as depicted in Fig. 5.b. Going deeper, the difference application should also take into account the new class `Line` by reproducing the same effects induced by $\Delta_{1,2}$ on the original `Circle` and `Polygon`.

In order to let the approach be general, this operation requires the definition of correspondences which identify the model elements upon which the modifications must be operated. Such correspondences are based on the concept of model weaving which has been successfully applied for metadata integration and evolution [12]. The specific definition of model weaving which is used here refers to the work in [2], where the authors proposed a model-based framework to establish model element correspondences in a generic way. In particular, it consists of the production of a weaving model $WM$ representing the mapping between the metamodels `LeftMM` and `RightMM`; like other models, this should conform to a specific weaving metamodel `WMM`.

Weaving links may be specified to relate also models instead of metamodels and this is the case of what is proposed in this paper in which weaving models are produced in order to relate input models with existing difference representations. Such correspondences are specified with respect to the `difference application` metamodel in Fig. 6. In particular, a difference application model (`DAModel`) consists of elements (`WElement`)
related through weaving links (WLink). According to the different kind of elements involved in weaving operations, InputWElement and DeltaWElement specialize the WElement concept. Moreover, ApplicationWLink is a specialization of WLink and relates elements of the input model (see the left role) with one of the difference representation (see the right role).

Fig. 5. Modified models

A sample difference application model is depicted in Fig. 7.c. The model is presented by using the tree-based editor of EMF 4 which is the framework that underpins the AMMA platform [13] used to implement the overall approach, as clarified in the following. Moreover, for the sake of illustration, the identifiers denoting the element correspondences have been replaced by dashed lines. The model in Fig. 7.c is only a fragment of the weaving specification that solves the problems stated above. In fact, the proposed application model discloses the possibility to apply the existing difference model \( \Delta_1, \Delta_2 \) in Fig. 4 to any model that conforms to the base metamodel \( MM \). In the example, the model \( M_1' \) of Fig. 5.a is taken into account in order to reuse and apply \( \Delta_1, \Delta_2 \) to produce the model \( M_2' \) of Fig. 5.b. In more detail, by means of the application model the modifications in \( \Delta_1, \Delta_2 \) are related to elements of \( M_1' \). For instance, the deletion of the Position class is specified by relating it (and the contained attributes) to the instances \( c7, datti, \) and \( datt2 \) in \( \Delta_1, \Delta_2 \) through the application links \( w1, w2, w3. \)

---

Fig. 7. Fragment of a sample difference application model

respectively. Moreover, the links \(w_6, w_7\) denote the deletion of the \textit{pos} attribute of the class \texttt{Polygon} and the introduction of the specialization with the new class \texttt{Figure}. The reader should note that the modification can be applied even though the class \texttt{Polygon} has a name different than the initial class \texttt{Polygons} in Fig. 3.a. This is feasible since the change expressed in \(\Delta_{1,2}\) by means of the instances \(c_3, att_1, c_4\) does not express any modification of the attribute \textit{name}, hence the \texttt{updatedElement} class will have the same name of the changed one. Interestingly, the new added class \texttt{Line} is also considered in the difference application. In fact, since the modifications that should be employed on it are the same as those for the class \texttt{Polygon}, the links \(w_4\) and \(w_5\) can be specified in order to use them for the class \texttt{Line}.

In general, the definition of difference application models can be supported by heuristics raising its automation level [14] like structural similarity which has proved to be an efficient element matching mechanism [9, 10]. For instance, in [10] the calculation starts by evaluating signature similarity, that is by comparing the combinations of type, kind and name of the given entities. Relations can be matched as well by means of their signatures plus those of linked source and target elements. If multiple valid candidates
exist, the mapping is refined through structural similarity computations. In our scenario, each time a mapping is discovered, it can be stored in the difference application model being specified. Then the automated weaving mechanism can be fine-tuned through manual intervention which could be required both to correct erroneous matchings and to establish custom mappings.

In order to apply differences with respect to the correspondences specified in weaving models, a new enhanced model transformation has to be generated by \textit{MMD2ATL}. In particular, the new \textit{MMD-MM2MM} can be applied to a source model \(M_1\) in order to obtain a target model \(M_2\) with respect to the differences specified in a model \(\Delta\) and the correspondences given in the difference application model \(DAM\). The transformation implements the rules to apply on a model \(M_1\) the addition, deletions and changes specified in the \(\Delta\) with respect to the application links specified in \(DAM\). In general, for each metaclass \(MC\) in the metamodel \(MM\), the transformation contains the following rules:

- \textit{AddedMC2MC}: for each element in \(\Delta\) that conforms to the \textit{AddedMC} metaclass the rule creates in \(M_2\) a new instance of \(MC\) setting the structural features according to the specification of the \textit{AddMC} element;
- \textit{ChangedMC2MC}: for each element in \(\Delta\) that conforms to the \textit{ChangedMC} metaclass the rule reads the links in \(DAM\) and verifies whether the change is applied to \(M_1\) or not. In positive cases the rule updates the changed elements in \(M_1\) according to the modifications specified in \(\Delta\);
- \textit{UnchangedMC2MC}: the rule copies the instances of \(MC\) which have to be the same both in \(M_1\) and \(M_2\). In particular, given an instance of \(MC\) in \(M_1\) the rule verifies whether there are application links in \(DAM\) which specify the deletion or the changing of the element. Only in negative cases the element is copied to \(M_2\);

As previously stated, the \textit{MMD-MM2MM} transformation can be generated by means of the \textit{MMD2ATL} higher-order transformation which takes as input the metamodel \(MM\) the models \(M_1\) and \(M_2\) conform to. Due to space limitation, the transformation is not provided here but it can be downloaded from [15].

4 Related work

Over the last years a number of approaches to model-based differencing of models have been proposed, e.g. [6, 9–11, 7] to mention a few. All of them can be considered symmetric deltas, according to the classification in [3]. The works in [6] and [7] differ from the others since they represent differences in a \textit{self-contained} way, i.e. they do not refer to any elements belonging to the base models. On the contrary, other works are \textit{relational} in the sense that they are based on weaving models [2] (or similar techniques) which contain correspondences between elements in the base models and thus cannot abstract from them. The main difference between these mechanisms is that the self-contained representations contain only information about added, deleted and changed elements disregarding those elements which are left unchanged. This can have an impact on the size of the difference models as witnessed by the coloring techniques (see [11] for a discussion on how to reduce difference model population). All the mentioned approaches may give place to difference application, but as a matter of facts only [6] and [7] provide support to it.
This paper aims at applying model differences as model patches. Thus, it generalizes the self-contained approach in [6] and, under limited assumptions, the one in [7] in order to apply differences also to models which are different from the base ones. In this respect, since the other approaches ([9–11]) are relational and not independent of the base models, they cannot be used as model patches. It is worth noting that there exist several alternatives for merging models, like for instance ReuseWare 5 and EML [16]. However, those approaches would require a new specification of difference application semantics for each metamodel taken into account. In fact, the difference application proposed in this work extends the already existing technique on model differencing presented in [6]; the previous difference application engine has been improved by adding the supporting knowledge coming from the weaving model to perform the desired manipulations on arbitrary input models. In this respect, the higher-order transformation exploits the basic difference metaelements semantics to derive the corresponding application engine tailored to the given metamodel.

Finally, patches have been introduced by Larry Wall in 1985 as a Unix utility. It updates text files according to instructions contained in a separate difference file with the possibility to relax context constraints by means of a fuzz factor. In our work the application of manipulations can be given a certain degree of fuzziness, the context of a difference is implicitly provided by its structure and the fuzz factor conveys the degree of structural similarity required for a modification to be operated. Other interesting proposals are Darcs [17] and Unison [18], text-based version management systems similar to CVS 6. One of the main distinctive features of Darcs is an underlying theory of patches, on which it is coherently based. The representation of a patch is defined as the manipulations induced by that patch if applied in a certain context. In this paper the representation of a delta is the set of changes caused by that document according to the input model and its matchings with difference elements. Concerning Unison, it is a file synchronizer whose job is to maintain consistency between replicated directory structures. Unison is endowed with different approaches for identifying filesystem differences. Unfortunately, these approaches are less appropriate for calculating the differences of models since they mainly rely on the Unix diff tool.

5 Conclusions and future work

Difference models can record the modifications a model undergoes during its life-cycle, as shown in [6]. By means of higher-order transformations it is possible to reproduce the modifications encoded in a difference model in order to reconstruct the newer version of a model starting from the old one. This paper presented an approach to generalize such construction and consider model differences as model patches, i.e. modifications which are formally described by a model and can be reproduced in any model conforming to the base metamodel. Such application is realized by coupling a difference model with a weaving model, that is a morphism between models, able to link the intended modifications to the model elements to be modified. Such morphism is called difference application model and can be defined i) by means of structural analysis, ii) in an implicit way, e.g. it can be derived directly from the persistent identifiers of the elements.

---

5 ReuseWare Website: http://www.reuseware.org
6 CVS Project Web site: http://www.nongnu.org/cvs
which are considered during the comparison, or iii) explicitly by the designer who intends to arbitrarily reproduce the modifications. In any case, at distinguished difference application models correspond different reproductions of the same modifications.

Efficient and reliable extraction of differences among two models opens up many possibilities for future work. Especially we are interested in an industrial validation of the presented approach, but in order to pursue this objective a preliminary analysis of the available model differencing tools is necessary.

References

Change Impact Analysis of Model-Driven Development Systems using Evolution Scenario Templates

Bedir Tekinerdogan 1, Ersin Er2

1 Bilkent University, Department of Computer Engineering, 06800 Bilkent Ankara, Turkey
bedir@cs.bilkent.edu.tr

2 Hacettepe University, Department of Computer Engineering, 06800 Beytepe, Ankara, Turkey
ersin.er@cs.hacettepe.edu.tr

Abstract. Model-Driven Software Development aims to support the development and evolution of software intensive systems using the basic concepts of model, metamodel and model transformation. In parallel with the ongoing academic research, MDSD is more and more applied in industrial practices. Like conventional non-MDSD practices, also MDSD systems are subject to changing requirements and have to cope with evolution. In this paper we provide a scenario-based approach for analyzing the impact of changes that apply to model-driven development systems. The approach defines so-called evolution scenario templates that can be used to describe specific evolution scenarios. The impact of each evolution scenario is measured as the required number of so-called evolution transformation patterns that are needed to evolve the system. A case study is used to show different kind of evolution scenarios and the required evolution transformation patterns.

Keywords: Change Impact Analysis, Evolution, Model-Driven Development

1. Introduction

In traditional, non-model-driven software development the link between the code and higher level design models is not formal but intentional. Required changes are usually addressed manually using the given modeling language. Because of the manual adaptation the maintenance effort is not optimal and as such sooner or later the design models become inconsistent with the code since changes are, in practice, defined at the code level. One of the key motivations for introducing model-driven software development (MDSD) is the need to reduce the maintenance effort and as such support evolution. MDSD aims to achieve this goal through defining models and metamodels as first class abstractions, and providing automated support using model transformations. For a given change requirement the code is not changed manually but automatically generated or regenerated, thereby substantially reducing maintenance effort. Further, because of the formal links between the models and the code the evolution of artefacts in the model-driven development process is synchronized. The link between the code and models is formal. In fact, there are only models, and as such,
‘the documentation is the code’. Research on MDSD is continuing to improve the expressiveness of the three key abstractions of model, metamodel and transformation [9]. As such even better and more automated support to cope with changing requirements and as such to provide reuse, portability, interoperability, and maintenance.

Because of the promising benefits for development and evolution, MDSD is more and more applied in industrial projects. Albeit, MDSD provides from one perspective better support for evolution, it also introduces new dimensions and challenges for software evolution [3]. The reason for this is that the required changes might not only remain within the code level and as such addressed by transformations. Like conventional code, models, metamodels and transformations might be subject to changing requirements and as such require to evolve in due time. Moreover, changes to the metamodels and transformations might render the terminal models invalid [10].

The software evolution problem in MDSD needs to address different challenges. One of the initial and key issues in considering evolution in MDSD is the impact of changes to the existing systems. Before discussing how to adapt the MDSD artefacts we need to know how to model the changes and how to define their impact. To address this problem we propose a scenario-based approach for analyzing the impact of changes that apply to model-driven development systems. For modeling the required changes we define the notion of so-called evolution scenario, which is defined as a description of the need for changes due to concerns of stakeholders. To describe evolution scenarios so-called evolution scenario templates are defined. Each evolution scenario will usually have an impact on the MDSD system and require changes to the models, metamodels or transformations. To quantify this impact we have defined so-called evolution transformation patterns that define the possible ways in which an MDSD system can evolve. The impact of each evolution scenario is measured as the required number of so-called evolution transformation patterns that are needed to evolve the systems. A case study is used to show different kind of evolution scenarios and the required evolution transformation patterns.

The remainder of the paper is organized as follows: In section 2 we describe evolution transformation patterns. In section 3 we present the scenario-based impact analysis process. In section 4 we present a case study that we apply the process. Section 5 provides the related work and finally we conclude in section 6.

2. Evolution Transformation Patterns

At the highest abstraction level software evolution implies the evolution of concerns in the system. A concern is defined as a general matter of interest that is held by some stakeholders. Evolution in MDSD can be defined as the change of stakeholder concerns and their impact on model elements. Concerns can be classified as functional or non-functional, and relate to any model element in the model-driven life cycle. Figure 1 shows the relation between stakeholder, concern and model element.
To be more concrete we should elaborate on the notion of model element. For this we adopt the megamodel of MDSD as depicted in Figure 2. The megamodel consists of three basic elements, *Model, Metamodel and Transformation*. We use the term Transformation to refer to both Transformation Definitions and actual Transformation Instances interchangeably. Each Transformation transforms one or more models into one or more models with respect to their metamodels. So a Transformation is defined depending on the source and target models’ metamodels as well as its own metamodel which it conforms to.

In principle each MDSD system can be considered as an instantiation of the megamodel and as such consist of a number of models, metamodels and transformations. In parallel with this evolution of MDSD systems can be considered as the evolution of these model elements. More concretely, a change requirement will require the adaption of one or more of these megamodel elements.

To formalize this evolution of the model elements we define the notion of *evolution transformation pattern* or *ETP* for short. An ETP can be defined as a possible evolution step in an MDSD system. Evolution is in principle defined as the addition, removal or update of a concern to model elements model, metamodel and transformation. We distinguish between *primitive ETP patterns* and *composite ETP patterns*. Table 1 defines the number of defined primitive evolution transformation patterns. Note that each pattern has a unique identifier starting with the letter p. An evolution step consists of a left part defining the initial state, followed by a hollow arrow defining the evolution action, and a right part which defines the state after the adopted changes. Patterns *p1 to p3* define the creation of new model, metamodel or transformation. Patterns *p4 to p6* define the removal of existing elements. Patterns *p7 to p9* define the update of the existing model elements. Finally, patterns *p10 to p12* define automatic generation of model elements.
Table 1. Primitive Evolution Transformation Patterns

<table>
<thead>
<tr>
<th>Id</th>
<th>Evolution Transformation Pattern</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1.</td>
<td>$\varepsilon \rightarrow M$</td>
<td>Model is created</td>
</tr>
<tr>
<td>p2.</td>
<td>$\varepsilon \rightarrow MM$</td>
<td>Metamodel is created</td>
</tr>
<tr>
<td>p3.</td>
<td>$\varepsilon \rightarrow T$</td>
<td>Transformation is created</td>
</tr>
<tr>
<td>p4.</td>
<td>$M \rightarrow \varepsilon$</td>
<td>Model is removed</td>
</tr>
<tr>
<td>p5.</td>
<td>$MM \rightarrow \varepsilon$</td>
<td>Metamodel is removed</td>
</tr>
<tr>
<td>p6.</td>
<td>$T \rightarrow \varepsilon$</td>
<td>Transformation is removed</td>
</tr>
<tr>
<td>p7.</td>
<td>$M \rightarrow M'$</td>
<td>Existing model is updated to model $M'$</td>
</tr>
<tr>
<td>p8.</td>
<td>$MM \rightarrow MM'$</td>
<td>Existing metamodel is updated to metamodel $MM'$</td>
</tr>
<tr>
<td>p9.</td>
<td>$T \rightarrow T'$</td>
<td>Existing transformation is updated to transformation $T'$</td>
</tr>
<tr>
<td>p10.</td>
<td>$M \xrightarrow{t} M'$</td>
<td>Existing model is automatically generated</td>
</tr>
<tr>
<td>p11.</td>
<td>$MM \xrightarrow{t} MM'$</td>
<td>Existing metamodel is automatically generated</td>
</tr>
<tr>
<td>p12.</td>
<td>$T \xrightarrow{t} T'$</td>
<td>Existing transformation is automatically generated</td>
</tr>
</tbody>
</table>

LEGEND: $M$ Model, $MM$ Meta-Model, $T$ Transformation model

Required changes could be defined as the execution of one or more of these nine patterns. To reflect more complex evolution steps composite ETPs can be defined from these primitive ETPs. Given the broad range of possibilities for defining an MDSD system, however, the set of possible composite ETPs is infinite and likewise it is difficult if not impossible to provide an exhaustive list of composite ETPs. Rather we can list some interesting composite patterns that might frequently occur. In Table 2, for example, we provide example composite ETPs that can be defined as a composition of the primitive ETPs. Hereby, the elements on the left that require a change are defined as graded elements. The elements which remain intact are not graded.

The first pattern in Table 2 defines the evolution of a model element that requires also evolution of the related metamodel. This pattern is typically a composition of the primitive ETPs $p7$ and $p8$. Updating an element means adding, removing or updating a sub-element. To define this additional information we parameterize the patterns using $p<op>$ whereby $op$ can be either add, remove or update. If no specific operation is given then $p<op>$ is used. For example an instance of pattern $p4$ could be either $p4<add>, p4<remove>$ or $p4<update>$. The first composite ETP can be defined as a
composition of p7<add> and p8<op>. Hereby p8<op> denotes that we need to update the metamodel but no specific action is given.

### Table 2. Example Composite Evolution Transformation Patterns

<table>
<thead>
<tr>
<th>Id</th>
<th>Evolution Pattern</th>
<th>Transformation</th>
<th>Explanation</th>
<th>Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><img src="image1" alt="Pattern 1" /></td>
<td>MM → MM'</td>
<td>New model element is added/removed or updated in model M, requiring also an update of metamodel</td>
<td>p7&lt;add&gt; ; p8&lt;op&gt;</td>
</tr>
<tr>
<td>2.</td>
<td><img src="image2" alt="Pattern 2" /></td>
<td>MM → MM'</td>
<td>Metamodel evolution requiring evolution of model</td>
<td>p8&lt;op&gt; ; p7&lt;op&gt;</td>
</tr>
<tr>
<td>3.</td>
<td><img src="image3" alt="Pattern 3" /></td>
<td>MM1 → MM2</td>
<td>Target metamodel is changed, requiring also a change of transformation. In addition target model will be changed (automatic transformation)</td>
<td>p8&lt;op&gt; ; p9&lt;op&gt; ; p10&lt;op&gt;</td>
</tr>
<tr>
<td>4.</td>
<td><img src="image4" alt="Pattern 4" /></td>
<td>MM1 → MM2</td>
<td>Source metamodel is changed, requiring a change of transformation, target metamodel and target model (automatic transformation)</td>
<td>p8&lt;op&gt; ; p9&lt;op&gt; ; p10&lt;op&gt;</td>
</tr>
</tbody>
</table>

The second pattern in Table 2 defines the case whereby the evolution of the metamodel requires adaptation of the conforming models. This pattern can be defined as consisting of p8<op> followed by p7<op>. The third pattern defines the change to a target metamodel that on its turn requires the adaptation of transformation definition and retransformation of the target model. As such the required primitive ETPs are p8<op> ; p9<op> ; p10<op>. Finally the fourth pattern describes the case in which the source
metamodel needs to be changed requiring on its turn the adaptation of the target metamodel, transformation definition and the automatic retransformation of the target metamodel. Obviously, as stated before several other patterns could be defined but due to space limitations we do not further elaborate on this.

3. Scenario-Based Analysis

In the previous section we have observed that evolution in MDSD relates to change of model elements, and the change of model elements might trigger the change of other related model elements. To define the possible changes we use the notion of evolution scenarios. Evolution scenarios, or usually called change scenarios have been widely discussed in the scenario-based architectural analysis domain [4]. Each evolution scenario defines the necessary change with respect to some concern for some stakeholder. In general, an exhaustive analysis of all evolution scenarios is neither possible nor necessary. Rather we will focus on the selected set of scenarios that are derived from key concerns as defined by stakeholders. Our goal hereby is twofold. First of all, we would like to record these scenarios, and likewise support the documentation of the evolution process. Second, we would like to analyze the impact of these concerns on the model driven system. The result of the analysis could support the model-driven engineer in adopting the right design decisions.

3.1 Process

The scenario-based analysis of model-driven systems consists of the following steps:

1. **Model the system.** First of all the MDSD system is modeled as consisting of a set of models, metamodels and transformations. After this, step conformance relations between models and metamodels, as well the transformation definitions and their relation to metamodels, need to be explicit.

2. **Describe evolution scenarios:** Concrete scenarios are derived that define the required changes that will be made to the MDSD system over time. Scenarios are described using so-called evolution scenario templates (ESTs) which will be explained in the next section.

3. **Define impact of evolution scenario.** Given the current state of the model-driven system the impact of each scenario is analyzed and described. This includes the selection of the required primitive ETPs as defined in Table 1.

4. **Do cost analysis based on scenario interactions:** For each model element it is checked how many scenarios impact the element. This will provide a global overview of the sensitivity of the elements in the MDSD system. The more scenarios are interacting in the element, the more sensitive it is to the given set of scenarios.

5. **Overall evaluation:** Finally, the identified evolution scenarios might be weighted in terms of their relative importance and this weighting is used to determine an overall ranking and as such the required effort for evolving the MDSD system.
3.2 Evolution Scenario Templates

The description of potential changes is represented using so-called evolution scenario templates or ESTs for short. The evolution scenario model that we adopt is given in Table 3. The evolution scenario template consists of two parts: a descriptive part that is provided as an input and a computed part, which is the output of the change impact analysis process. The descriptive part provides an Id, an optional stakeholder field, and a description of the evolution scenario; the computed part provides the results of the change impact analysis that consist of Required Pattern Instantiations which indicates the required primitive ETP instantiations. The field Change Scope Diagram provides a diagram in which the required evolution step is defined.

<table>
<thead>
<tr>
<th>Evolution Scenario Id:</th>
<th>A unique id for identifying evolution scenarios</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>Detailed description of the change scenario in natural language</td>
</tr>
<tr>
<td>Stakeholder: (optional)</td>
<td>The actor that requires or is interested in the change scenario</td>
</tr>
<tr>
<td>Required Pattern Instantiations:</td>
<td>The required evolution transformation pattern</td>
</tr>
<tr>
<td>Change Scope Diagram:</td>
<td>Illustrates the diagram together with the updated model elements</td>
</tr>
</tbody>
</table>

4. Example Case

In this section we will illustrate the impact analysis process for analyzing a simple case for University Information System. Although the example is relatively small it is considered to be enough for our purpose and it utilizes most of the patterns mentioned. In section 4.1 we will describe the model of the system. In section 4.2 the evolution scenarios are described with the required ETP instantiations. Finally in section 4.3 we will describe scenario interactions for each element.

4.1 Describing the MDSD system

A University Information System deals with students, courses, academic stuff, enrollments and various academic processes in a university. Figure 1 shows a model of the MDSD system which consists of the metamodels Entities_MM and Java_MM, the transformation Entities_to_Java which is simply denoted as T, and the models Entities_M and Java_M. The metamodel Entities_MM constitutes a subset of UML class package without generalization relationship and any behavioral features. The metamodel Java_MM defines an elaborate Java metamodel. The transformation is used to transform the model Entities_M to a model Java_M. A small part of the model Entities_M is also illustrated on the lower part of the figure.
Figure 3. Simple model for a small part of a University Information System.

4.2 Describing Evolution Scenarios and Evolution Transformation Patterns

In the following we will describe three different evolution scenarios as described in Table 4. The first scenario requires adding Instructor as a separate new entity to the Entities model. Previously instructor of a course was modeled with a string attribute in the Course entity for representing his/her name. A new Instructor entity as well as an association relationship with the Course entity should be added to the system. The existing attribute of the Course entity should be removed. The scenario itself does not have any impact on metamodels or transformation.

The second scenario requires adding GraduateStudent as a new entity as a specialization of the existing Student entity. This requirement triggers the need for generalization as a reusability mechanism. Entities metamodel does not support generalization in its current state. So before adding GraduateStudent, Generalization Relationship should be added to the Entities metamodel. Introducing new model elements in the Entities metamodel also requires updating the Entities to Java model transformation for handling the new model elements.

The third scenario requires persisting some selecting entities. To save space we have illustrated only the end result in the Change Scope Diagram part. In order to be able to specify which entities should be persisted and how the persistence will take place, the notion of persistence should be introduced into the system. For introducing persistence into the system the Entities metamodel can be updated in order to be able to specify which entities need to be persisted. A new metamodel needs to be added for abstracting the Persistence Framework. For simplicity we consider a general relational persistence framework which can be modeled with a single metamodel. A new transformation from the updated Entities metamodel to the new Relational Persistence Framework metamodel should also be added. Finally Entities model should be updated for specifying which entities need to be persisted.
<table>
<thead>
<tr>
<th>Evolution Scenario Id:</th>
<th>S1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>Store personal information about Instructors</td>
</tr>
<tr>
<td>Required Pattern</td>
<td>p7&lt;add&gt;(Entities_M); p7&lt;remove&gt;(Entities_M); p7&lt;add&gt;(Entities_M); p10(Java_M);</td>
</tr>
<tr>
<td>Instantiations:</td>
<td>Add Instructor entity. Remove instructor attribute from Course. Add Course-Instructor association. Generate Java model.</td>
</tr>
<tr>
<td>Change Scope Diagram:</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scenario Id:</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>A new GraduateStudent entity as a specialization of Student entity should be added by firstly introducing Generalization into the system</td>
</tr>
<tr>
<td>Required Pattern</td>
<td>p8&lt;add&gt;(Entities_MM); p9&lt;add&gt;(Entities_to_Java); p7&lt;add&gt;(Entities_M); p10(Java_M);</td>
</tr>
<tr>
<td>Instantiations:</td>
<td>Add Generalization relationship to Entities metamodel. Add rule that handle Generalization into Entities to Java transformation. Add GraduateStudent entity to Entities model. Add Generalization between Student and GraduateStudent entities. Generate Java model.</td>
</tr>
<tr>
<td>Change Scope Diagram:</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scenario Id:</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>Selected entities in the system should be stored to a relational database</td>
</tr>
<tr>
<td>Required Pattern</td>
<td>p8&lt;add&gt;(Entities_MM); p2(RPF_MM); p3(Entities_to_RPF); p7&lt;add&gt;(Entities_M); p10(RPF_M);</td>
</tr>
<tr>
<td>Instantiations:</td>
<td>Add persistence related information to Entities metamodel. Add Relational Persistence Framework (RPF) Metamodel. Add Entities to RPF metamodel. Add persistence related information to Entities model. Generate RPF model.</td>
</tr>
<tr>
<td>Change Scope Diagram:</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>
4.3 Cost Analysis

After the impact analysis of each scenario we can now look at each model element separately and define the number of scenarios that interact at these elements. Scenarios interact at an element in case it requires a change to the corresponding element. In our initial system we have two metamodels and two models that conform to these metamodels. Each scenario changes the system in various dimensions so that the existing model elements are updated; removed or new model elements are added.

In addition each scenario requires a set of ETP instantiations. Although the number of ETPs involved in a scenario can coarsely serve as a metric of cost, a more detailed cost analysis can be performed by taking the characteristics of each pattern into account. ETPs have different effects on model elements. Intuitively, we can state that changing a metamodel has a higher cost than changing a model. So the more a scenario interact with metamodels the more cost that scenario yield to. For example, S1 in Table 4 does not interact with any metamodel, where both S2 and S3 interact with one metamodel. This makes the cost of S2 and S3 higher than S1's. The same approach can be expanded to cover transformations as well as detailed operations performed in each pattern. For example, updating a sub-model element (be it belong to either a model, metamodel or transformation) can be considered as cheaper than adding a new sub-model element.

5. Related Work

Model-Driven Software Development (MDSD) has been proposed mainly to solve the complexity in software development by increasing abstraction level and to provide more controlled means to handling evolution which is a major challenge since the early days of software engineering. While MDSD can be good at dealing with changing requirements, it has been stated that using MDSD brings more dimensions to consider with respect to evolution [3]. In this context, researchers especially have studied co-evolution of models and metamodels [6][8][10].

In this paper we observed the impact of several evolution scenarios on a model-driven system from a broader perspective. For similar purposes the concept of megamodel has been proposed as a means to model and reason about a model-driven system completely [7]. In [2] authors have provided an impact analysis for UML models in particular.

Scenario-based analysis approaches have been widely applied and validated over the past several years in the software architecture design community. Several scenario-based architecture analysis methods have been developed each focusing on particular quality attributes [4]. In general, scenario-based analysis methods take as input a model of the architecture and measure the impact of the predefined scenarios on it in order to identify the potential risks and the sensitive points of the architecture. A scenario is generally considered to be a brief description of some anticipated or desired use of the system [1]. Hereby, it is implicitly assumed that scenarios correspond to the particular quality attributes that need to be analyzed. The concept of Evolution Scenario
Template is as such builds on this existing work. The important issue in ESTs is that it not only describes the scenario but also maps to evolution transformation patterns.

6. Conclusions

In this paper we have provided a systematic approach for analyzing the impact of evolution scenarios on a given MDSD system. The approach defines the so-called evolution scenario templates that can be used to define concrete evolution scenario. Each evolution scenario maps to a number of evolution transformation patterns, thereby defining the required impact for the evolution scenario. We have adopted the approach for a rather small case study including only a few set of elements, however, we think that the same mechanism can be applied to a larger case study. The impact analysis consisting of a set of recorded evolution scenarios is important from two perspectives. First of all, we have now a simple but effective means to document the evolution scenarios. Second, the impact analysis results can be used to guide the maintenance process because it defines the necessary patterns for adapting the system. Our future work will include a larger case study and the definition of case tools to support the overall process.

References

Service Oriented Architecture Definition Using Composition of Business–Driven Fragments

Sébastien Mosser, Mireille Blay–Fornarino, and Michel Riveill

University of Nice – Sophia Antipolis
CNRS, I3S Laboratory, MODALIS & RAINBOW Teams
Sophia Antipolis, FRANCE
{mosser,blay,riveill}@polytech.unice.fr

Abstract. Services Oriented Architecture are built through the composition of services (e.g. Web Services) to define complex business process (e.g. Orchestrations). Well known methodologies focus on identifying services and orchestrations at design time. However the orchestration design phase is still a heavy burden, as it induces to deal with both technical and business domain concerns. This article proposes to use an evolution framework (Adore) to capitalize architects knowledge and best practices into “evolutions”. Architects can build business-driven orchestrations by composing reusable “evolutions” following a design–by–composition approach. We apply this approach to build a legacy SOA called SEDUITE (validation platform for the French national research project FAROS).

1 Introduction

Web Services and Orchestrations [1] provide a way to implement scalable SOA (Services Oriented Architecture, [2]) under the WSOA acronym (Web Services Oriented Architecture). Architects define Web Services to publish elementary services. Complex business processes are defined through orchestrations which assemble elementary functionalities into a control–flow. The W3C defines orchestrations as “the pattern of interactions that a Web Service agent must follow in order to achieve its goal” [3]. Orchestrations are the keystones of a WSOA, as they represent the business-driven processes. Existing methodologies (e.g. SOMA [4]) focus on process identification. When using these methods, architects identify business services and orchestrations at the model level. Existing high–level formalisms and languages are used to express these processes (e.g. BPMN [5], BPEL [6]). But the process definition must be written from scratch as code. A business–driven step–wise development implies to know what happened in previous steps before modifying the legacy software in a coherent way: when defining different solutions corresponding to different customers of a same business domain, each architect will then use its own approach to define solutions which are then difficult to compare and even more to make evolve.

The contribution of this paper is to show how a business architect can design a WSOA by composing reusable process fragments. It proposes a business point of view of the software evolution problematic, focusing on high level evolution
expression. Performing an evolution in our context means enhancing an existing business process, adapting it to fit customers expectations. We are positioning our work at design time: an architect wants to design a business-process, so he/she will retrieve a design model of an existing business-process and adapt it to his/her customer needs.

Where Aspect Oriented Programming (AOP, see sec. 5) targets on reusable aspect identification and weaving, this work focuses on the composition of evolution at the model level and is presented as a complementary approach. We use as running example (described in sec. 2) a reference WSOA called SEDUITE. Section 3 describes the ADORE evolution framework and its benefits for SEDUITE case study. Section 4 exposes validation of this work on the whole legacy system, and section 5 discusses related work. Finally, section 6 concludes this paper by showing some exciting perspectives of this work.

2 Running Example: the SEDUITE system

SEDUITE is an information system designed to fit academic institution needs. It supports information broadcasting from academic partners (e.g. transport network, school restaurant) to several devices (e.g. user’s smart-phone, PDA, desktop, public screen). This system is built upon a WSOA and used as a validation platform by the FAROS project\(^1\). The implementation follows WSOA methodological guidelines [7], positioning experimentations as a typical usage of a WSOA. The system is deployed inside two institutions. Further information about implementation and partners can be found on the project web site\(^2\).

The SEDUITE system defines two kinds of business processes: (i) sources of information to retrieve information from partners and (ii) providers to assemble sources invocations according to broadcast policies. Designing such processes is a business-driven task, as they must fit with school users needs. We identify inside the SEDUITE system a set of chronic situations where the business architect reuse a set of good practices inside system’s orchestrations. We present in this section a relevant subset of chronic evolutions identified inside SEDUITE.

When working on sources of information, we identified four chronic evolutions: (i) cache to implement cache mechanism when a source can slow down a process (e.g. external partner, temporary server crash) (ii) shuffle to shuffle a set of explicitly unordered information (e.g. a set of scrapped pictures), (iii) toInfo to transform an arbitrary data into one of type Information using an XSL meta-transformation and (iv) truncate to restrict the cardinality of an information set to a given size. When defining (or enhancing an existing one) a provider, we identify three evolutions: (i) addSource to add a new source of information, (ii) dry to dry up a source when time is over and (iii) multiCalls to allow a source to be called several times (e.g. a device broadcasting weather data for several cities instead of a single one).

\(^1\) http://www.lifl.fr/faros
\(^2\) http://www.jseduite.org
Performing these *evolutions* inside the system relies on the architect knowledge of those patterns. Even if those chronic *evolutions* are well documented, it is unrealistic to believe that everybody will follow the guidelines (SEDUI TE is developed by ten different stake-holders, on three different continents). It rapidly results into an unmaintainable system where in front of identical situations, each architect leaves its own footprint on the architecture to propose an equivalent solution.

3 Composition of Orchestration Fragments

ADORE (*Activity moDel supOrting oRchestration Evolution*) is a platform designed to support orchestration evolutions. A dedicated meta–model supports the definition of orchestrations and orchestration fragments at a higher abstract level than usual formalism (*e.g.* BPEL). The representation used by ADORE supports the definition of a composition algorithm able to weave fragments into base orchestrations.

Orchestrations defined as BPEL code (based on the ADORE language restriction explained in next section) can be transformed into their representations conforming to the ADORE meta–model. The reciprocal transformation generates BPEL entities from an ADORE representation (using a technique similar to topological sort) to reach legacy servers infrastructure.

3.1 The ADORE Meta–Model

In ADORE, an orchestration of services is defined as a partially ordered set of activities. The different types of activities that can be defined in ADORE are a subset of the types of activities defined in the BPEL industrial specifications. These include (*i*) service invocation (denoted by *invoke*), (*ii*) variable assignment (*assign*), (*iii*) fault reporting (*throw*), (*iv*) message reception (*receive*), (*v*) response sending (*reply*), and (*vi*) the null activity, which is used for synchronization purpose (*nop*). In an ADORE process model, each process starts with a *receive* activity and ends with a *reply* activity.

An activity is identified by an unique identifier. Activities can use zero or more inputs and outputs. Unlike UML activity diagrams, in which an activity can have a nested structure, an ADORE activity is always primitive. A more complete description of the ADORE meta–model can be found on the project web site.

We also define a graphical syntax to represent ADORE orchestrations. A box represents an activity inside the orchestration, and an arrow between two boxes means that the targeted activity of the arrow is allowed to start at the end of the source one. A label on an arrow represent a condition. Figure 2 represent a simple provider (which aggregates internal news and restaurant menu) using both formalisms (graphical and tuples).

---

3 http://www.adore-design.org
Fig. 1: ADORE meta–model (formal & simplified class–diagram)

(a) ADORE graphical notation for $p$

```
p \equiv \{ \{ a_1, a_{20}, a_{21}, a_3, a_5 \},
                  \{ a_1 \lessdot a_{20}, a_1 \lessdot a_{21}, \ldots, a_3 \lessdot a_5 \}, \emptyset \}

a_1 \equiv (a_1, \text{receive}, \emptyset, \emptyset)

a_{20} \equiv (a_{20}, \text{invoke} (\text{internalNews}, \text{getAll}),
                  \emptyset, \text{news})

a_{21} \equiv (a_{21}, \text{invoke} (\text{menu}, \text{getToday}),
                  \emptyset, \text{menu})

a_3 \equiv (a_3, \text{assign} (\text{concatenate}),
                  \{ \text{news}, \text{menu} \}, \{ \text{result} \})

a_4 \equiv (a_4, \text{reply}, \{ \text{result} \}, \emptyset)
```

Fig. 2: ADORE example: $p$, a SEDUITE provider
3.2 Expressing Business-Driven Fragments Using ADORE

Defining an evolution in this context means to define a process fragment which enhances the behavior of an existing process activity. The fragment can interact with the (eventually unknown) targeted activity \( t_a \) in several ways, using three special activities: (i) the hook activity reifies \( t_a \) in the evolution context (providing an access to input and output variables), (ii) an activity \( P \) representing all immediate predecessors of \( t_a \) and (iii) an activity \( S \) representing all immediate successors of \( t_a \). Fig. 3 represents two evolutions using our graphical syntax: (i) the AddCache evolution used inside SEDUIE to avoid slow response time and (ii) the Dry evolution used to dry up a source after a given hour (e.g. lunch menu is useless after lunch time).

![Diagram](a) \( E_1: \text{AddCache} \) best-practice  
(b) \( E_2: \text{Dry} \) best-practice

Fig. 3: Expressing best-practices as ADORE evolutions

To integrate an evolution into an existing process we unify hook, \( P \) and \( S \) activities with \( t_a \) and its related activities. The unification is defined as a set of substitution \( \sigma \) [8] which perform modifications on the legacy process. A conflict detection algorithm can detect incoherent structure inside the builded process (e.g. concurrent write accesses to a shared variable) and ask for complementary knowledge to solve conflict. The resulting process is then transformed into a concrete BPEL process and then deployed inside an application server.
3.3 Evolution Composition Algorithm

When an architect wants to apply a set of evolutions \( E \equiv \{e_1, \ldots, e_n\} \) on a given targeted activities (e.g. an information source should be cached, truncated and shuffled), evolutions must be composed before being integrated into the targeted process. As a full description of this algorithm is out of the scope of this paper (see [9]), we propose an informal and textual description: to perform the composition of behavioral evolutions, the algorithm starts to unify all hook, \( P \) and \( S \) activities inside \( E \) elements. Then, it propagates guards relations (conditional executions of activities) on the built evolution. It results in a single and new evolution \( e \equiv \text{Merge}(E) \). Fig. 4 illustrates the merge of the two previously described evolutions. The conflict detection mechanisms is executed over this process to detect incoherent structures. The resulting process can be integrated into an existing orchestration using the previously described technique.

![Diagram](image)

Fig. 4: \( E' \equiv \text{Merge}(\{E_d, E_c\}) \)

4 Validation & Implementation

This section presents some experiments using the approach and then describes the implementation of the platform. As SEDUITE is a new platform with a
medium complexity size, we plan to perform further validations on a more complex system. Grid computing community is a good field of experience as it intensively relies on domain–dedicated data–flow to express intensive computations.

**SEDUITE Case Study Results:** We use this approach to model the two complete SEDUITE systems. In this paper, we focus on POLYTECH’SOPHIA system. POLYTECH’SOPHIA SEDUITE software is composed by 15 elementary Web Services implemented in JAVA (representing more than 8 KLoC), six BPEL orchestrations (almost 2 KLoC and 14 KLoC of WSDL and XSD artefacts) and two providers (more than 2 KLoC and 11 KLoC of artefacts).

Table 1 shows the usage of the identified source evolutions into SEDUITE orchestration. For each orchestration, it indicates which evolution were applied, and the cardinality of the activity set $A^*$ before (resp. after) the merge process. The $\Delta_f$ column represents the growth factor of the activity set. The average $\Delta_f$ is 1.61, with a maximum of 2 for the FeedReader orchestration. This result shows that up to 50% of a process can be defined as the composition of evolution in this context.

| Orchestration | Evolutions | $|A^*|$ | before | after | $\Delta_f$ |
|---------------|------------|-------|--------|-------|--------|
| BusLocalizer  | AddCache   | x     | 5      | 6     | 1.2    |
| RssFeedReader | x          | x     | 4      | 8     | 2      |
| ImageScraper  | x          | x     | 5      | 8     | 1.6    |
| RestaurantMenu| x          |       | 6      | 10    | 1.7    |
| PictureAlbums | x          | x     | 9      | 15    | 1.7    |
| TvShows       | x          |       | 4      | 5     | 1.3    |
| Weather       | x          |       | 5      | 9     | 1.8    |

Table 1: Evolution usage in SEDUITE orchestrations

Table 2 shows the usage of evolutions to define a given provider: SchoolProvider. This provider is the more complex defined inside SEDUITE as the information set retrieved by this provider is broadcasted to a public screen in the main hall (so it handles an important set of sources). It aggregates ten different sources, each of them was added using a specialization of the AddSource evolution. For ten sources, only three do not use evolutions to enrich their information handling logic. The seven others use one or two evolutions to customize the information

---

4 As it is more complex than the other one.
5 $1 KLoC \equiv 1000$ lines of relevant code.
6 These sources (InternalNews, StudentConvocation & Weather) broadcast dedicated information. As a consequence, they are defined as internal and dedicated services, so they do not need any others adaptation.
retrieval process, which show the coverage of the method. The final provider is composed of 60 activities, where 58 (≈ 96%) are automatically added and ordered.

<table>
<thead>
<tr>
<th>Source</th>
<th>AddCache</th>
<th>DryUp</th>
<th>MultiCalls</th>
<th>Shuffle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absences</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BreakingNews</td>
<td></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>BusLocalizer</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Calendar</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ImageScraper</td>
<td>×</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>InternalNews</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Menu</td>
<td></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>StudentConvocation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TvShows</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weather</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: SchoolProvider & Evolutions

**Implementation** SEDUITE reference implementation\(^7\) uses JAX-WS Web Services as elementary bricks and BPEL 2.0 orchestrations to define complex business processes. The ADORE metamodel is implemented following the Model Driven Engineering (MDE) paradigm, using ECLIPSE EMF [10] as technological layer. A bridge between ADORE and an industrial BPEL metamodel (Wtp project) is defined through a model transformation written using the KERMETA language [11]. As merge algorithms are defined in an inductive way and rely on logical properties, we use the PROLOG language to implement and validate them. Porting these algorithms into an object oriented language as JAVA or KERMETA (to bundle them in an evolution dedicated graphical modeling tool) is an ongoing work.

### 5 Related Work

Douence defines Aspect Oriented Programming (AOP) in [12] as “*a set of language mechanisms which enable the introduction of non anticipated functionalities in a base application*”. The AO4BPEL engine [13] brings aspect-oriented functionalities to an existing orchestration server. An aspect is supposed to be automatically woven on a given code without any additional information. When multiples aspects must be woven at the same place (so called shared join points [14]) existing solutions work at the code level (e.g. adding an order between two

\(^7\) [http://www.jseduite.org](http://www.jseduite.org)
aspects). Interactions between aspects (i.e. conflicts in ADORE terms) should then be managed at the level of the aspect definition, directly inside the advice code. Development by evolution composition is different, as we focus on the composition problem when a set of evolution must be applied at the same point. The composition algorithm is based on composition of a set of evolutions and doesn’t depend of any order. Conflict detection steps ensure that the evolutions can be composed and that the build orchestration is valid.

Our work can also be compared with AOP works on mixins and traits [15]. However when these approaches rely on a composition based on multiple inheritance and code injection, evolution composition is based on the composition of graphs of activities and can generate new activities and relations.

Previously described approaches are code–driven. To help business experts and let them define business processes, higher level mechanisms and formalism were described and normalized. The business process community made several attempts to handle evolution and variability of these processes, like generic workflows [16] or process families [17]. These techniques put the focus on the representation of such concerns, without addressing the composition of business processes problematic. This issued is solved in ADORE through the composition algorithm.

6 Conclusions

In this article, we explain how an evolution framework called ADORE can be used to capitalize architects knowledge. This knowledge is reified as evolutions and then composed to build new architectures. The methodology is illustrated on a medium-complexity existing software called SEDUITE. This software is a validation platform of the FAROS research project (French national consortium) which aims to build reliable SOA through the composition of evolutions. Further validation of the methodology will be done on grid–computing data–flow for medical imaging.

Composition algorithm always focus on enriching an existing orchestration. We never address the evolution retract problem in an incremental approach. As far as we are in ADORE implementation, retracting an evolution means re–computing the global result from scratch instead of reasoning on the delta introduced by this evolution retract. An immediate perspective is to enrich evolution algorithms to take care of these kinds of considerations.

For now, the proposed methodology is designed for software architects. But when a business domain is restricted to its pure essence (CIM following MDE vocabulary), it should be manipulated by users who are not computer scientists but business experts. A long term perspective is to capture the information broadcasting domain into a dedicated meta–model and then write model transformation that reaches the proposed methodology at the PrM level. This abstraction will allow SEDUITE end user (e.g. schools’ headmaster) to build their own dedicated providers without knowing anything about the underlying system: they will just express their sources set and handling policies at CIM level, and the
composition algorithm will do the composition at PtM level before generating ready-to-deploy BPEL orchestrations.

Acknowledgments This project is partially funded by the French Research Agency (ANR) through the FAROS project. The ADORE framework is one of the platforms targeted by FAROS. The SEDUITE software (through its reference implementation) is used as one of the validation platform.

References

State-based vs. Operation-based Change Tracking

Maximilian Koegel, Markus Herrmannsdoerfer, Jonas Helming, and Yang Li

Institut für Informatik
Technische Universität München
Boltzmannstrasse 3, 87548 Garching
{koegel, herrmama, helming, liya}@in.tum.de

Abstract. In recent years, models are increasingly used throughout the entire lifecycle in software engineering projects. In effect, the need for managing these models in terms of change tracking and versioning emerged. However, many researchers have shown that existing approaches for Version Control (VC) do not work well on graph-like models, and therefore proposed alternative techniques and methods. They can be categorized into two different classes: state-based and operation-based approaches. Existing research shows advantages of operation-based over state-based approaches in selected use cases. However, there are no results available on the advantages of operation-based approaches in the most common use case of a VC system: review and understand change. In this paper, we present and discuss both approaches and their use cases. Moreover, we present a design of an empirical study to compare a state-based with an operation-based approach in the use case of reviewing and understanding change.

1 Introduction

Today, models are an essential artifact throughout the entire lifecycle in software engineering projects. Model-driven development is putting even more emphasis on models, since they are not only an abstraction of the system under development, but the system is (partly) generated from its models. Consequently, models are in use throughout the development process from requirements over design to deployment, including management of the process itself. With the adoption of model-driven development in industry, the need for managing these models in terms of change tracking and versioning emerged. Version Control (VC) is already in wide-spread use for textual artifacts such as source code.

However, many publications, e.g. [1–7], recognized that existing VC approaches do not work well on models, which essentially are attributed graphs. The traditional VC systems are geared towards supporting textual artifacts such as source code, managing them on a line-oriented level. In contrast, many software engineering artifacts including models are not managed on a line-oriented level, and thus a line-oriented change management is not adequate. For example, adding an association between two classes in a UML class diagram is a structural
change, which is neither line-oriented, nor can be managed in a line-oriented way. A single structural change in the diagram is managed as multiple line changes by traditional VC systems. 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 [1]. Different approaches have been proposed to cope with the shortcomings of existing methods and techniques to better support change tracking and versioning of graph-based models. They can be categorized into two different classes: state-based and change-based approaches [8].

**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 [8]. This activity is often referred to as *diffing*. The diffing process can be viewed as a calculation to derive the change post-mortem, and is generally expensive in computation time.

**Change-based approaches** record the changes, while they occur, and store them in a repository. There is no need for *diffing*, 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 [8]. The recorded operations can be applied to a state to transform it to the successor state.

Several publications exist that show advantages of change-based and in particular operation-based approaches over state-based approaches in use cases such as conflict detection and merging [7, 9, 10], consistency management [11], repository mining [12], and coupled evolution [13]. However, there are no results available on the advantages of operation-based approaches in the most common use case of a VC system: reviewing and understanding change. We claim that understanding change is the most important use case of a VC system, as it is required for almost any other use case, e.g. commit, update, merge, etc. Therefore, we believe it is essential to conduct experiments on how well this use case is supported by the state-based and operation-based approaches.

In this paper, we discuss representatives of the different approaches as well as the advantages and disadvantages of each type of approach in general. To compare them, we present frequent use cases of a VC system, and show how they are supported by the respective approach. Finally, we present the design of an empirical study we intend to conduct to compare a state-based with an operation-based approach for the use case of reviewing and understanding change.

**Outline.** Section 2 introduces common use cases of a VC system. Sections 3 and 4 introduce the state-based approach and operation-based approach, respectively. Section 5 presents the design of the empirical study, and Section 6 concludes the paper with a short summary.

## 2 Use Cases for a Version Control System

A VC system has to fulfill a lot of use cases, many of which do not differ from a state-based to a change-based system. Consider the use case of baselining, in which the user marks a certain approved version (e.g. a release). Since changes
are not at all considered in this use case, both approaches appear identical from a user’s point of view. Consequently, we only focus on use cases where a difference between state-based and change-based systems arises. We derived these use cases from well-known tools, such as the Revision Control System (RCS) [14], the Concurrent Versioning System (CVS) [15], and Subversion (SVN) [16], from research tools such as SiDiff [17] and UNICASE [18], and from publications such as [8] and [19]. Figure 1 illustrates the use cases, that we consider important for discussing the differences between state-based and change-based change tracking. For every use case, we provide a short name, and give a description:

**Update.** The user retrieves changes between her local version and a target version (mostly the current head version) from the repository. These incoming changes can be reviewed by the user, before they are incorporated into the local working copy of the model. If the user accepts the changes and they do not conflict with local changes, the version of the local working copy is set to the target version, and the changes are incorporated into the local copy.

**Commit.** The user decides to share the changes from her local working copy with the repository. The user reviews the changes before the commit to ensure that only intentional changes are sent to the repository. If the user proceeds, the changes are sent to the repository to create a new version. In case they conflict with other commits that occurred since the last update, the commit is canceled by the VC system and an update must occur first.

**Merge.** When incoming changes conflict with existing local changes in the update use case, the merge use case is initiated. The goal of the use case is to filter or transform the incoming and/or local changes, until they no longer conflict. The result will be incorporated into the local workspace. The merging process involves manual work in most cases, requiring to review the changes. Merging may also occur if two branches in the repository are synchronized or rejoined, which essentially requires the same steps.

**Fig. 1.** Use cases of a VC system (UML use case diagram)
Blame. To find out how and by whom a problem or an inconsistency was created, the user is interested in finding recent changes on a certain model part. Typically, the last $n$ changes on a selected set of model elements need to be retrieved. The user reviews these changes to find the causing change.

Show History. The user (often a project manager) reviews the history to get an impression on the current activity in a project. Mostly, she is not interested in individual changes, but in an overview of how many and which type of change on how many artifacts occurred.

Show Differences. The user is interested in reviewing the differences between two versions of a model, for example two releases. The two versions are typically not very close in terms of the number of changes between them.

Revert. The user wants to undo some changes in the local working copy. To ensure that the right changes are undone, she reviews the changes before.

Review and Understand Changes. The user reviews changes to understand what was changed and most importantly how it was changed.

Interestingly, the first seven use cases are placed most prominently in many VC systems and their clients [15, 16, 18]. We claim this is due to the fact that these are the most frequently executed use cases for the majority of users. In all of these seven use cases, the user is reviewing changes in one or another way. As is shown in Figure 1, all use cases thus include the use case ”Review and Understand Changes”.

3 State-based Change Tracking

State-based approaches derive differences by comparing two states, e.g. a version and its successor, after the changes occurred. This activity is often referred to as diffing, and is performed in two phases: matching and comparison. In the matching phase, for each node in one state, the corresponding node in the other state is found. The matching can be based on the similarity of the node’s content or on the graph structure it is connected to [20]. If the model supports unique identifiers, the matching can be found in $O(1)$, otherwise $O(n^2)$ are required for $n$ nodes in a model [21, 22]. Chawathe et al. even claim that the matching problem for two states is NP-hard in its full generality [23]. In the comparison phase, each node is compared with its matching partner from the other state to derive changes if there are any. The comparison calculation requires $O(n)$. The space complexity for the whole diffing process is $2n$, since both states need to be present. Change is not a first level concept in a state-based system. The diffing process can be viewed as a calculation to derive the changes post-mortem.

Since the VC system must not be able to observe the changes, while they occur, a total separation of the modeling tools and the VC system is possible. This is a clear advantage over change-based systems. It is even possible to use line-oriented VC system, and to perform diffing on the client side. For example, EMF Compare [24] is a diffing tool for EMF (Eclipse Modeling Framework) [25] models. There are three main disadvantages of the diffing concept: (1) The time
order of changes is lost, and it cannot be perfectly derived. For understanding changes, the time order might be important. Moreover, the time order is useful for conflict detection and merging [9]. (2) Groupings of changes to composite changes are lost. Refactoring operations e.g. cause many changes that can be grouped. This reduces the number of changes, and represents the change at a higher level of abstraction. Deriving composite changes, e.g. to detect refactorings, is difficult and in some cases even impossible due to masking problems [26]. (3) The computational complexity for diffing is high, especially if changes between many states need to be derived, or the model is of a large size [21, 22]. We suspect that disadvantages (1) and (2) will reduce the ability of users to understand change, and hope to be able to show this in the empirical study.

Considering the use cases presented in Section 2, we can make the following observations for the state-based approach:

**Merge.** The merge result can not be as accurate, as composite changes are not available [10, 9]. Refactoring operations for example might only be partly reflected, if not all their caused changes are accepted.

**Show History.** The computational complexity for diffing could result in a severe performance problem, if looking at many versions and the changes that occurred in between them, since diffing is required for every version.

**Review and Understand Changes.** Disadvantages (1) and (2) are impacting the ability of humans to understand change. The lost time order of the changes could help to understand the context in which the changes were performed. Composite changes could group many changes that look unrelated, and thus have to be grouped in the user’s mental model.

## 4 Operation-based Change Tracking

In contrast to state-based approaches, change-based approaches record the changes, *while* they occur. This implies that change-based systems persist changes in a way that reflects how they were actually performed. There is no need for diffing, since the changes are already available by design. Operation-based approaches are a special class of change-based approaches which represent the changes as transformation operations on a state [9]. An operation can be applied to a state to transform it to the successor state [8]. Figure 2 shows the simplified taxonomy of operations from the UNICASE system [4, 18]. All operations refer to one ModelElement that is being changed by the operation, and that is unambiguously identified by a unique identifier. An AttributeOperation changes the value of an attribute for a model element. A ReferenceOperation creates or removes one or more links between model elements. A CreateDeleteOperation creates or deletes a model element. A CompositeOperation allows to group several related operations.

The change-based approaches have one disadvantage in common: they require the VC system to be present when the changes occur, i.e. when the modeling tool is manipulating the model. This requires an integration of the VC system into the modeling tool. However, this does not imply that the system must
instrument the modeling tool, but can only use the infrastructure on which
the tool is built. For change recording, observer mechanisms can be used, and
for composite detection, the command pattern can be used. In case of EMF
models, one can rely on the EMF notifications and command stack [25]. This
effectively decouples the VC system from the modeling tool. In general, change-
based approaches can preserve the time order in which the changes occurred.
This is an important information for understanding changes, but is also useful
for other applications such as conflict detection and merging [10, 9]. Moreover,
the exact times at which the changes occurred can be recorded. Operation-
based systems can record composite operations which express the fact that the
contained operations occurred in a common context. For example, a refactoring
can be captured in a composite operation. This can help to understand changes,
but also in conflict detection and merging [10]. Robbes et al. even claim that
only an operation-based VC system allows for effective research on evolution,
since it provides all the required information [12]. Operation-based systems can
provide a method to canonize a sequence of operations, which hides operations
that are fully masked by later operations. For example, a **Pull up to Superclass**
refactoring is fully masked by a later deletion of all the involved classes.

Considering the use cases presented in Section 2, we can make the following
observations for the operation-based approach:

**Update.** The operations incoming from the repository are presented to the user.
In case the difference between local and target version is large, the system
canonize the operations to get a more compact representation. Conflict
detection can fully rely on the operations, their time order and composites to
supply a more accurate result and avoid unnecessary conflicts [10]. In general,
conflict detectors apply a conservative estimation: If they are unsure about
a potential conflict, they raise a conflict to avoid later data corruption.

**Commit.** The recorded operations can be presented to the user, possibly after
a canonization. No diffing is required. Conflict detection can again profit
from the additional information.

**Merge.** The merge can operate on the top level operations. Therefore, less
decisions are needed, since many operations are contained in a composite
operation. Moreover, the decisions can not partially mask a refactoring as
opposed to the state-based case [26].

---

**Fig. 2.** Taxonomy of operations (UML class diagram)
State-based vs. Operation-based Change Tracking

Show Differences. This use case is best served by a state-based representation. It can be implemented directly by relying on an existing state-based approach or by deriving it from the recorded operations.

Review and Understand Changes. The changes can be shown as operations in the correct time order and grouped as composite operations.

The operation-based approach might seem very different from a state-based approach, but in effect it is only an enhancement. It records additional information that is lost in the state-based approach. In an operation-based approach, we can perform everything that can be done in a state-based approach, by just ignoring the additional information. This boils down to the question whether the additional effort for recording the changes is justified by its advantages. We claim that this is the case, and we hope to find statistical evidence in our empirical study that operation-based change tracking improves the user’s ability to review and understand change.

5 Design of the Empirical Study

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

1. Do users better understand the changes in a state-based or an operation-based representation? The metrics for understanding are the taken time, self-assessment of users and assessment by interviewers.

2. Which factors influence the user in understanding a state-based or an operation-based representation? Candidate factors are number of changes, kinds of changes and dependencies among changes.

5.1 Setup

We chose representatives for the two different approaches to change tracking.

State-based change tracking is represented by the open-source tool EMF Compare [24] which is the state-of-the-art diffing and merging implementation for EMF. As we do not want to disadvantage EMF Compare by design, the used matching strategy is based on unique identifiers to ensure correct matchings.

Operation-based change tracking is represented by two open-source tools from different domains in order to be more representative: UNICASE for model evolution, and COPE for metamodel evolution.

UNICASE is a CASE tool based on a unified model [18]. The unified model covers the whole development process from requirements over design to deployment, including management of the process itself and its artifacts. UNICASE consists of a modeling tool to author instances of the unified model, and a central repository to persist them including VC. UNICASE is implemented based on EMF, and realizes operation-based change-tracking, conflict detection and merging [4].

COPE is an EMF-based tool to automate the migration of models in response to metamodel adaptation [27, 13]. COPE records the operations carried out on
a metamodel, and allows to attach information on how to migrate models. To significantly reduce effort, COPE allows to reuse generic couples of metamodel adaptation and model migration. The recorded information can be used to automatically migrate models to the newest version of the metamodel.

5.2 Input

Both UNICASE and COPE were used to record operation histories which we use as an input to the empirical study.

UNICASE was employed in a project named DOLLI2 (Distributed Online Logistics and Location Infrastructure 2) at a major European airport. The objective of DOLLI2 was integrating facility management and telemetry data into the tracking and locating infrastructure developed in the previous project, together with expanding the 3D visualization on desktop computers as well as porting it to mobile devices. More than 20 developers worked on the project for about five months. This resulted in a comprehensive project model consisting of about 1000 model elements and a history of over 600 versions.

COPE was employed to reverse engineer the evolution of the metamodels from the Graphical Modeling Framework (GMF) [28]. GMF allows to define a graphical editor by models from which editor code can be automatically generated. The metamodels consist of about 1500 metamodel elements, and the history of over 100 versions.

5.3 Execution

We apply the following systematic process to execute the empirical study:

1. Choose Users. We choose 10 users which are familiar with the input, as well as 10 users which are not familiar with the input.

2. Extract operation-based representation. For each chosen user, we extract 20 commits from the operation-based histories. Each commit consists of a sequence of operations. To be able to correlate the answers with the complexity of the contained operations, we randomly sample 5 commits for each of the following categories: (1) contains only AttributeOperations, (2) contains also ReferenceOperations, (3) contains also CreateDeleteOperations, and (4) contains also CompositeOperations (for the taxonomy, see Figure 2).

3. Generate state-based representation. We generate the state-based representation for all the sample commits using EMF Compare.

4. Question Users. We present the commits to the user in a state-based or an operation-based representation. Each user should not receive both representations for the same commit, as the first representation might ease the understanding of the second representation. However, every user should be shown almost the same amount of either operation-based or state-based representation. The user should try to understand the changes. We assess the understanding of the user by means of three metrics: (1) the taken time, (2) the self-assessment by user, and (3) assessment by interviewer. Metric (1)
provides a quantitative answer, whereas metrics (2) and (3) provide qualitative answers. For the qualitative metrics, we use a scale with the following five values: very difficult (=1), difficult (=2), OK (=3), easy (=4), very easy (=5).

5.4 Evaluation

We perform a number of statistical tests to evaluate the measurements. To show the soundness of the measurements, we determine the following correlation:

**Self- vs. interviewer assessment.** Do the users really understand the changes? The study failed, if both assessments do not vary in a similar way. We carry out a t-test of the null hypothesis, \( H_0 : \mu_1 = \mu_2 \), on self-assessments and interviewer assessments, where \( \mu_1 \) and \( \mu_2 \) are the mean values of the two assessments, respectively.

To answer research question 1, we determine the following correlations:

**Self-assessment on state-based vs. operation-based.** Is it easier to understand the changes in a state-based or an operation-based representation? We carry out a t-test of the null hypothesis, \( H_0 : \mu_1 \leq \mu_2 \), on both self-assessments, where \( \mu_1 \) and \( \mu_2 \) are the mean values of the two assessments, respectively.

**Time on state-based vs. operation-based.** Can changes be understood faster in a state-based or an operation-based representation? We conduct a t-test similar to the previous one.

To answer research question 2, we determine the following correlations:

**Self-assessment on state-based vs. commit size, self-assessment on operation-based vs. commit size.** Is it more difficult to understand the changes, if more operations are involved in a commit? We perform a regression analysis on the assessments, in case the assessments vary in a certain way with increasing commit size.

**Self-assessment on state-based vs. operation category, self-assessment on operation-based vs. operation category.** Is it more difficult to understand the changes, if the contained operations are more complex? For each \( i \in \{1, 2, 3\} \), we perform a t-test of the null hypothesis, \( H_0 : \mu_i \geq \mu_{i+1} \), on the assessments of operation category \((i)\) and \((i + 1)\), where \( \mu_i \) are the mean values of the assessments of the operation category \((i)\).

6 Conclusion and Future Work

We reviewed both state-based and operation-based approaches for change tracking. We compared both approaches by means of typical use cases of VC systems. We are convinced that operation-based approaches are superior to state-based approaches with respect to the use case of reviewing and understanding changes.

To compare both approaches, we will conduct an empirical study as a next step. We expect to observe, that in most cases, the operation-based representation is better suited to understand changes than the state-based representation. Moreover, we want to find out how different factors like e.g. the number of changes influence the activity of understanding changes.
References

Using change intentions to guide evolution and versioning in Model Driven Development

Rafael Leaño, Rubby Casallas and Nicolás López

Universidad de los Andes, Bogotá, Colombia
{r-leano, rcasalla, ni-lopez}@uniandes.edu.co

Abstract. Model Driven Development (MDD) provides a solution to meet industry requirements for software quality and development speed. MDD uses models as first-class entities in software development, and in this scenario, models are an active part of a development process and also of its evolution. Our experiences show that tracking model changes and maintaining and evolving generated and manual code are fundamental activities in the evolution of MDD implementations. Updating generated code is easy; however, automatically updating manually written code is unfeasible. In this paper we present a model versioning strategy where model analysts can characterize the intentions of changes; intentions are high level descriptions of the reasons and impacts of changes made in the models. We track intentions to the specific impacted code and assist developers in the process of maintaining the code. Our strategy helps to avoid inconsistencies caused by model changes by using the intention of changes as a first-class entity which guides model versioning and eases maintenance processes.

Keywords: model evolution, model versioning, model transformation, model comparison, Model-Driven Development (MDD), software evolution.

1 Introduction

Model Driven Development (MDD) is a paradigm that aims at increasing the quality and speed of development using models, model transformations and code generation templates as first class entities [1]. Model Driven Architecture (MDA) is the Object Management Group's (OMG) flavor for MDD. In MDA an initial model is created with a high level of abstraction and no implementation details, this model is the platform independent model (PIM) and it is transformed and refined adding more implementation details in possibly several steps. Eventually the level of detail is enough to consider it a platform specific model (PSM) which is finally used to generate software assets, mainly code [2].

Although it is possible to generate 100% of the code based on the source models, this is not really feasible in practice [3]. For this reason, generated code must be completed manually by the developers, thus, we can categorize code of an application as generated code (code generated by the MDD approach) or manual code (code written by the developers). We must consider these two different kinds of code to be
Using a Platform Independent Delta (PID)

able to coherently evolve the software as a whole. It is often that manual code relies and depends on the generated code it extends.

Software has to evolve; in this paper we are interested in a specific type of evolution; changes introduced in the PIM as a result of evolution of requirements which have an impact both in generated and manual code. Manual code can become obsolete as a result of changes in the PIM, whereas the generated code is updated by the regeneration process. Finding manual code impacted by the change, identifying what part of it becomes obsolete and updating it are major challenges for the evolution of an MDD implementation.

In this paper we present a strategy called Platform Independent Delta (PID) where change intentions guide the maintenance process. In our strategy, we allow the analyst in charge of modifying the PIM to specify the intentions and impact of the changes made in the PIM. We track these changes and their intentions in a Delta Model (DM) which maintains versioning information. The DM is the main source of information that aids the developer in the maintenance process of manual code. We trace the changes into the specific parts of the source code that are impacted. We provide the developer, within the development environment, with visual aids that indicate which parts of the manual code are related to the model changes, and thus are possibly obsolete. The developer has direct access to the high level intentions that guide her in the process of making the changes necessary to evolve the manual code coherently with the evolution of the PIM.

The structure of this paper is as follows: in section 2 we introduce the specific MDD approach in which we will apply the PID strategy along with an example application generated using the line. In section 3 we further detail the challenges of maintaining manual code and model changes as the MDD implementation evolves. Afterwards section 4 will explain in detail the PID strategy and then we analyze briefly some related work in section 5. Finally section 6 presents our conclusions of the work done and possible future work to continue this project.

2 Our Case Study

The MDD approach we used is based on a Model Transformation Chain (MTC) developed by the Software Construction Group at Universidad de los Andes. This MTC is used to generate Management Information System (MIS) applications with CRUD requirements and JEE5 as the generation platform. An initial PIM is eventually transformed into a PSM following a series of transformations shown in Fig. 1.
2.1 University Example

We introduce a sample application that we created using the MTC. The application manages students, courses and faculties for a university; its initial model (EA model) is shown partially in Fig. 2. With this model and the MTC we generate partially completed code. The developers must afterwards complete this generated code with manual code, this manual code relates strongly to the generated code and any change in the generated code can disrupt the manual code. The university application models the concepts of Student and University, amongst many more not shown here. We present only a subset of the details of a student: name and age. The generated code for the Student class will have the name and age as attributes.

![Fig. 2. University example](image)

3 Problem: Evolution

As an MDD implementation grows, models, metamodels and transformations become part of the evolution process. We are interested particularly in a situation where the MTC is relatively stable and most of the evolution requirements are related to modifications in the high level, business models, in the MDD implementation.

Maintaining the generated and manual code is a complex challenge. In this context there are several reasons why we need to track the changes made in the EA model and the intention behind those changes:

![Fig. 3. Introducing changes in the university application](image)

In our university example, as a result of evolution, we have decided to update the model from version 1.0 to version 2.0. For this update, we want to change the model to store separately a student's name from her last name, so we added an attribute `lastname` to the EA model. This also means now that the `name` attribute should only contain the first name of the student. We also decided to change the `age` attribute with the `birthday` attribute, removing the need to update the student's age each year. Fig. 3 shows the changes we performed on the EA. If we consider that the university and student concepts are only a small part of the model then manually searching for
Using a Platform Independent Delta (PID)

differences between generated and manually coded lines tangled in hundreds of files that represent the two versions of the model is a time-consuming and error-prone task.

Additionally, previously to our model changes, the developers added code manually for the sample university application. We cannot lose this code when we regenerate with the updated model, this is why code generation frameworks like Acceleo\(^1\), MOFScript\(^2\) and Xpand\(^3\) offer protected regions where code can be written and will be kept intact after the generation process. As the developers use these regions to fill in manual code, the code is embedded and tangled with the generated code, which has negative implications: First, manual code can only be inserted where these regions exist and, even though protected regions can be added anywhere, if we need to add new regions we must modify the templates of the MTC. Second, using protected regions makes it more difficult to find manual code since it is left scattered all around different files along with the generated code.

If manual and generated source code reside in a single file the file is no longer disposable (it cannot be erased at will) or the manual code will be lost along with the generated code [4]. It also makes synchronization more difficult. Imagine this case: In the university example developer A makes the changes mentioned previously to the model and developer B makes changes to the Address concept (not related to the other changes) and both add manual code to the generated code (in different classes). When they synchronize the repository will find all files as modified (since code generation regenerates every file, not only the changed ones). This application could easily contain more than 200 classes and the developers in charge of synchronizing would have to manually find the files that were really changed and merge them or manual code would be lost.

![Fig. 4. Types of obsolete manual code: Syntactically incorrect and semantically incorrect](image)

After regenerating code one last problem remains. Manual code can no longer be correct as a result of the changes in the model, particularly if the code references the parts that changed in the model. Searching and identifying which manual code has to be updated can be easy if the changes on the generated code result in compilation errors in the manual code. However there are cases (especially when adding concepts to the model) where manual code remains computationally correct but semantically incorrect.

In the university example, two manual methods were added when the model was in version 1.0; `canDrink()` and `getFullName()` (Fig. 4). When we regenerate using model version 2.0 the `canDrink()` method is marked by the compiler as syntactically

---

1 http://www.acceleo.org/pages/home/en
2 http://www.eclipse.org/gmt/mofscript/
3 http://www.bar54.de/benjamin.klattXpand.pdf or http://www.openarchitectureware.org/
incorrect since the age attribute is not present anymore. A developer can easily find this problem and correct it, on the other hand, the method getFullName() has no compilation error and will not be marked, thus making it harder to pinpoint. Nonetheless the method is logically incorrect since it does not include the student's last name. Finding and correcting code that becomes logically incorrect is a very complex task as the application grows and manual code gets buried deeper within generated code, becoming a big flaw concerning MDD implementations’ evolution.

Once developers find obsolete code, they must update it correctly according to the changes in the model. So understanding why the analysts introduced a change gives the developer great guidance regarding how should she should modify the code. Unfortunately developers do not always have this information available.

4 Our Strategy: A Platform Independent Delta

In this section we introduce the Platform Independent Delta (PID) strategy which addresses the problems of MDD implementation evolution. In our strategy, see Fig. 5, we include as part of the SPL an additional model which we call the Delta Model (DM). With the DM the business analyst, in charge of evolving the PIM, can characterize the modifications between two versions and the intentions of these changes. We trace this information, by means of code analysis, directly to the code structures related to each of the changes. With these intentions, the developers can understand the impact the changes have on manually implemented code. The developers can then make the necessary modifications to the code to maintain the functionality of the application and consistently evolve the manual code. To implement this strategy we first had to modify our MTC to create a clear division of the manual code from the generated code (Fig. 5 num 1). We then present the main activities involved in our strategy: (Fig. 5 num 2) versioning the models, where the analyst can clearly model the changes and their intentions; (Fig. 5 num 3) searching for code that is possibly obsolete using the DM and a code analysis tool; and (Fig. 5 num 4) assisting the developer in evolving this manual code by tracing the change intentions to the delta model from source code structures.

4.1 Separating manual and generated code

We effectively separate manual code from generated code at development time by using virtual classes. Virtual classes are classes that are not considered at deployment
Using a Platform Independent Delta (PID)

time; they are only accessible during development. A pair of virtual classes represents a concrete class that will be deployed; each pair is made up of a generated virtual class and a manual virtual class. We merge these classes into one single complete class which will be the one deployed in the application, virtual classes are not accessible at runtime.

In the university example this translates to have a Student_gen class which is generated and a Student_man class with all the manual code. Both files will be merged to make up the Student class containing both codes in a single class. This way we can safely maintain generated files outside of version control and only synchronize the manual files. Using this strategy assures us the separation of manual and generated code without forcing any pattern on the source code.

We use a tool called JMerge \[12\] \[13\] from the Eclipse EMF and the JET projects to merge the code of the virtual files. JMerge creates an Abstract Syntax Trees for both files and merges them before rewriting the final code. We use the final code to compile the final version of the application. During development code is seen separated at all times. The rest of our strategy depends on this separation.

4.2 Versioning Models

This activity allows the analyst to compare two versions of the PIM and indicate the intentions of the changes. This activity starts when we have two versions of the PIM, such as version 1.0 and 2.0 described in our university example. We use a model comparison framework \[14\] to generate a model that identifies the differences between the different versions. This model is a first step in completing the Delta Model (DM) and specifies the elements added or removed from one version to the other. However, to complete the DM we need more information from the business analysts to fill in some blanks; specifically they can create relationships amongst various modifications and comment on them. These comments describe the intention of a change. An intention explains why a change was introduced and is the cornerstone to add value to the versioning process and to guide developers when evolving manual code to maintain consistency with the changes. The relation between changes shown by the comparison framework is fully understood by these intentions, for example a removal in the v1.0 model and an addition in the v2.0 model could mean a concept was relocated or simply renamed. Since intentions are the work of a human analyst they can be done so that any model change can be handled by them.

When we use the comparison framework in the University example we create a model that indicates that the age attribute was deleted while the lastname and
Using a Platform Independent Delta (PID)

*birthday* attributes were added. Only the business analyst knows that the removal of the *age attribute* and the addition of the *birthday* attribute are related, and that the *lastname* attribute in version 2.0 is related to the *name* attribute in version 1.0. The analyst then groups these changes (e.g., the *lastname* and *birthday* changes) into a single change and specifies the intention of each grouped changes. Fig. 5 shows the final DM in our example; the model specifies only two changes with their intentions.

After the analyst completes it, the DM becomes a first class entity in the evolution process. The DM is the means for our model version control and contains in a single, reduced file all the information we need to identify what was modified and why. Even if the comparison framework only detects additions and removals the analyst can relate those changes to handle all kind of modifications (concepts that are split, relocated, merged, etc.)

4.3 Searching Obsolete Code

We have argued that the biggest challenge when dealing with evolution of manual code in an MDD implementation is identifying code which is logically incorrect after the PIM changes. With the DM at hand we include it in our MTC and tag and propagate the changed concepts and use a code analysis tool to find the code that is potentially obsolete as shown in Fig. 7.

![Fig. 7. Tag and propagate concepts related to a change defined in the DM](image)

With the modified concepts tagged in each refined model we can generate a different set of artifacts designed for the code analysis tool to search the manual code for these modified concepts. With these artifacts we can identify that, for example, the *Student_man* class is referencing to the *name* attribute which is tagged since it’s related to the *NameChange* modification in the DM. The analysis only considers the manual partial classes since the generated code will never be obsolete as it is updated by the MTC in the generation process.

The code analysis tool we use is based on the Spoon [15] project. With this tool we can create Spoonlets which are extensions to the Java compiler to make additional validations on Java code. We use Spoonlets to visually mark the code related to the changes defined in the DM. In our example this means that our MTC will generate Spoonlets that will analyze the code in search for references to the *age* (AgeSpoonlet) and *name* (NameSpoonlet) attributes and point them as potentially obsolete code. Fig. 8 presents the visual cues that the developer will see in her IDE.
4.4 Evolving Manual Code

As we pointed out previously, not all of the manual code that references the changes is necessarily out of date. Fig. 8a shows a method that references the name attribute and is now obsolete; the getFullName() method now must also include the lastname attribute. Fig. 8b shows a method that also references the name attribute; this method however, is not obsolete. So how can developers know if the changes of the model have an impact on this manual code? Any attempt to automatically identify if a manually codified method is now out-of-date is futile since this code can be very complex and is related to functionality which is not described in the PIM (if it was possible to describe it in the PIM the functionality would be part of the generated code). So the best alternative we have is to provide the developer with as much information possible related to the intention of the change for her to be able to make well informed decisions and modify only the parts of the code that need changes.

Once developers find potentially obsolete code it is up to them to decide if the code must be updated, and, if changing the code, they must also ensure that the new code is updated correctly. To achieve this we use a traceability framework [16] to bring the intentions of the DM directly to the developers.

In the university example as the Spoonlet marks the potentially obsolete code in getFullName() it will show the relevant intention message for the modification regarding the name attribute in the DM. With this information the developer can easily identify that the getFullName() method is now obsolete, whereas the getFirstLetter() is not obsolete.

5 Related Work

We will now review some alternatives that could be used to solve some of the problems we tackled with the PID Strategy. We briefly valuate them and explain the reasons that lead us to the tools we used for the strategy.

The FUUT-je project (Fantastic, Unique UML Tool for the Java Environment) [8] shows a common solution for keeping manual code apart. FUUT-je integrates manual code in the models so that in the regeneration process, the manual changes are also inserted along with the generated code. This solution does not solve the model versioning problem and by having code inside the PID is tied to a specific language. Our PID Strategy does not involve changes in the PIM and developers are able to find manual code as source code, not as text attached to a model.

Markus Völter and Jorn Bettin suggest the use of pattern or tricks available in the target language to separate manual code from generated code [7] [9]. The patterns
and tricks all aim at having one file with manual code and one with generated code and relate them somehow. The problem with this is that we need to force the MDD implementation to use these patterns and tricks or code will not be separated. This is troublesome if the MDD implementation requires the use of special patterns that cannot coexist with the code separation pattern. Our solution does not interfere with the patterns chosen for the MDD implementation.

EMFCompare [9] is a part of Eclipse’s EMF Project and a solution to track model changes. It allows the comparison of models aimed at on-the-fly comparisons showing differences with the goal of synching and merging models to resolve conflicts in a repository. However we do not need a synchronized version of a model nor are we trying to solve the differences of two models. EMFCompare is a work in progress and is still an incubating project at the time we were writing this paper.

Finally, JastAddJ [11] extends the Java compiler to add validations, it is build on top of the Java 1.4 compiler and replaces it completely. JastAddJ would allow us to implement additional validations to java code to help us find obsolete code. When Java5 came an extension was added to incorporate annotations and such to the validations. Unfortunately JastAddJ is not integrated with the Eclipse IDE and extensions must be written in the JastAddJ Specification Language, also, completely replacing the Java compiler means that if code is written in a new version of java (i.e. Java1.6) the JastAddJ compiler becomes obsolete and can no longer be used. All this made us prefer the Spoon framework.

6 Conclusions

The PID strategy enhances the MDD approach allowing faster development and maintenance by reducing the gap between model developers and source code developers. The inside workings of the MTC do not matter (transformations and inner models) since developers can easily go back and forth from the problem domain to the solution domain. The PID strategy reduces maintenance time by preventing mistakes in code synchronization that result in lost code (and hence time and work lost), but mostly aids manual code evolution in: (1) Eliminates the time spend manually searching for obsolete code. (2) Obsolete code can no longer be overlooked. (3) Reduces greatly debug times since if manual code is successfully updated runtime exceptions are less likely to appear.

Aside from adding versioning information to the MDD implementations the PID strategy introduces the concept of intention to the changes by adding design decisions to the DM and using it as another input in the MTC for analyzing manual code and for informing developers not only where obsolete code might be but also the design reasons that lead to a change to help them evaluate and decide if code must be updated and the right way to update it.

The limitations of the comparison framework are removed with the use of intentions to relate simple changes like additions or removals to handle all kinds of modifications.

PID Strategy is an excellent complement to the MDD approach and applying it to an MTC benefits all the products derived from it. Further work can be done by better analyzing the DM and suggesting fixes for the incorrect code or applying basic fixes
using a platform independent delta (PID) automatically. The PID strategy is a forward-engineering strategy that could be used together with other strategies to boost even more its benefits.

The PID Strategy was validated with the MTC developed at the Universidad de los Andes introduced in chapter 2, further information on it can be found in [14] [15]

7 References

15. Barais, O.: SpoonEMF, une brique logicielle pour l'utilisation de l'IDM dans le cadre de la reingénierie de programmes Java5. In : Journées sur l’Ingénierie Dirigée par les Modèles (IDM), Lille, Nord-Pas de Calais, France (June 2006)