Configuration management for models: Generic methods for model comparison and model co-evolution

Z. Protić

This work has been carried out as part of the FALCON project under the responsibility of the Embedded Systems Institute with Vanderlande Industries as the industrial partner. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute (BSIK03021) program.

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA Dissertation Series 2011-12

A catalogue record is available from the Eindhoven University of Technology Library ISBN: 978-90-386-2650-5

Reproduction: Universiteitsdrukkerij Technische Universiteit Eindhoven

Configuration management for models: Generic methods for model comparison and model co-evolution

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op maandag 3 oktober 2011 om 14.00 uur

door

Protić Zvezdan
geboren te Novi Sad, Servië

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. M.G.J. van den Brand

Copromotor:
dr.ir. T. Verhoeff

Preface

> One thing only I know, and that is that I know nothing.
>
> *Socrates (469 BC–399 BC)*

Searching for the truth has been a privilege of mankind since the dawn of history. However, the truth has proven to be elusive, and not even the greatest minds of mankind could say that they knew the truth. Quite contrary, most of the philosophers will agree that the more one knows about the truth, the more questions about the truth one has. This situation was reflected in science— we still do not know the truth, but each new discovery brings us closer to the truth, and also raises new questions about the truth.

In this dissertation I present the results of my search for the truth. However, while the presented results answer selected research questions as truthfully as possible, as a result of this research many more questions are raised, that are to be answered by those who follow in my footsteps.

I would like to thank Mark van dan Brand, and the Falcon interview team, for giving me the opportunity to work on the research presented in this dissertation. I would also like to thank Marcel van Amstel for his unending support, both in terms of research related to the Falcon project, and in terms of adapting to the Dutch society. Next, I would like to thank my second supervisor, Tom Verhoeff,

for the fruitful philosophical discussions, which taught me to slow down my pace, and reflect on my work more. Moreover, I would like to thank both Mark and Tom for their guidance in improving my writing and communication skills. Furthermore, I would like to thank the reading committee – Gerti Kappel, Koos Rooda, and René Krikhaar – for reading and assessing this dissertation. I would also like to thank Luc, Jeroen, Arjan, Yanja, and Loek, and the Falcon team members, for the many hours of useful discussions. Furthermore, I would like to thank Zhare, Dragan and Jasen for accepting me as a friend since my first days in the Netherlands, and also to Natasha, Meri and Biba for being such good friends during these four years. Finally, I would like to thank my wife Sonja - without her ability to make me focus this dissertation would not have been written.

Summary

It is an undeniable fact that software plays an important role in our lives. We use the software to play our music, to check our e-mail, or even to help us to drive our car. Thus, the quality of software directly influences the quality of our lives. However, the traditional Software Engineering paradigm is not able to cope with the increasing demands in quantity and quality of produced software. Thus, a new paradigm of Model Driven Software Engineering (MDSE) is quickly gaining ground.

MDSE promises to solve some of the problems of traditional Software Engineering (SE) by raising the level of abstraction. Thus, MDSE proposes the use of models and model transformations, instead of textual program files used in traditional SE, as means of producing software. The models are usually graph-based, and are built by using graphical notations - i.e. the models are represented diagrammatically. The advantages of using graphical models over text files are numerous, for example it is usually easier to deduce the relations between different model elements in their diagrammatic form, thus reducing the possibility of defects during the production of the software. Furthermore, formal model transformations can be used to produce different kinds of artifacts from models in all stages of software production. For example, artifacts that can be used as input for model checkers or simulation tools can be produced. This enables the checking or simulation of software products in the early phases of development,

which further reduces the probability of defects in the final software product.

However, methods and techniques to support MDSE are still not mature enough. In particular methods and techniques for model configuration management (MCM) are still in development, and no generic MCM system exists. In this dissertation, I describe my research which was focused on developing methods and techniques to support generic model configuration management. In particular, during my research, I focused on developing methods and techniques for supporting model evolution and model co-evolution. Described methods and techniques are generic and are suitable for a state-based approach to model configuration management.

In order to support the model evolution, I developed methods for the representation, calculation, and visualization of state-based model differences. Unlike in previously published research, where these three aspects of model differences are dealt with in separation, in my research all these three aspects are integrated. Thus, the result of model differences calculation algorithm is in the format which is described by my research on model differences representation. The same representation format of model differences is used as a basis of my approach to differences visualization. It is important to notice that the developed representation format for model differences is metamodel independent, and thus is generic, i.e., it can be used to represent differences between all graph-based models.

Model co-evolution is a term that describes the problem of adapting models when their metamodels evolve. My solution to this problem has three steps. In the first step a special metamodel is introduced (a metamodel for metamodels - MMfMM). Unlike in traditional approaches, where metamodels are represented as instances of a metametamodel, in my approach the metamodels are represented by models which are instances of the MMfMM. In the second step, since metamodels are represented by models, previously defined methods and techniques for model evolution are reused to represent and calculate the metamodel differences. In the final step I define an algorithm that uses the calculated metamodel differences to adapt models conforming to the evolved metamodel. In order to validate my approaches to model evolution and model co-evolution, I have developed a tool for model evolution, and a tool for model co-evolution. These tools, together with small case-studies, are also described.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The pervasiveness of software is undeniable. From small personal gadgets like mobile phones or portable music players, via home appliances like television sets or DVD players, to large industrial machines like package handling systems or palletizers, software has found a way into most devices around us. However, surveys show that traditional software development methodologies are unable to cope with the size and scope of projects currently in industry [35, 37]. One reason for this is that the transfer of knowledge between different phases of the development process is problematic. Design decisions that have been made in one phase need to be manually interpreted in the next. For example, the detailed layout of a warehouse is given to software engineers who need to develop the software for controlling the package handling system in that warehouse. Since the development teams among which knowledge has to be transferred possibly have different backgrounds, this may lead to all kinds of *misinterpretations*, which, in turn, lead to defects in software.

There are two reasons for this problem. First, when the development phases involve different formalisms, there can be differences in semantics and expressive power of those formalisms. Therefore, it may occur that concepts expressed in one formalism cannot be expressed in the other. In an attempt to bridge these semantic gaps, a slightly different interpretation may have to be chosen for certain concepts. Second, design decisions tend to be insufficiently documented. For example, decisions that are considered to be trivial for a development team may have been omitted from the documentation. In this case, developers in a subsequent step may interpret these "trivialities" in a different way than intended. Surveys show that these problems result in situations where maintenance of the software accounts for up to 90% of the total cost of the software [46].

Model-driven software engineering (MDSE) is an emerging software engineering discipline intended to improve traditional software engineering (SE). MDSE aims at dealing with increasing software complexity and improving productivity [80, 103]. This is achieved by providing means to raise the level of abstraction from the problem domain rather into the solution domain, and to increase the level of automation in the development process. Raising the level of abstraction is achieved by employing domain-specific modeling languages (DSMLs). DSMLs offer, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, the particular problem domain [117]. Thus, a DSML enables system modelers to model in terms of the domain concepts rather than concepts provided by general purpose formalisms, which typically do not provide the required or correct abstractions. For example, in order to describe a package handling system, a modeler of such a system would be able to use a set of concepts such as workstations, conveyors, and storages, instead of generic concepts such as a UML [98] class. Thus, the set of domain-specific concepts would allow the modeler to express himself in the most natural way for that particular domain. Moreover, the concepts used in DSMLs are commonly expressed by using graphical primitives, which also improves understanding of the modeled system by the diverse set of stakeholders.

Automating the transition between different development phases is achieved by using model transformations, they provide a mechanism to automatically generate (or update) new models from existing models. This facilitates the transfer of models between the different phases of the software development life-cycle,

while ensuring consistency between the models. Moreover, by using model transformations, models do not have to be interpreted by humans, which, in combination with the domain-specific abstractions, greatly diminishes the risk of misinterpretations. Furthermore, since the process is automated, it is quicker and less error-prone.

Model transformations are also used for automated generation of various artifacts from models throughout the development process [101]. The resulting artifacts can be used as a starting point for the application of techniques such as model checking, model verification or model validation, in all phases of the development process, increasing the confidence of system modelers in the correctness of the final product.

However, the increase in productivity offered by MDSE has its price; unlike in traditional SE, where any textual editor could be used to create software, in MDSE, tool support is imperative. There are two main reasons for this.

The first reason is that both models, and model transformations, must be formally defined and tracked throughout the development process. The formal syntax of a model is described by using a *metamodel*, and the formal syntax of a model transformation is described by using a model transformation language. In traditional SE, the tracking of developed artifacts is done through software configuration management systems, and in MDSE it is done through model configuration management systems. However, while the process of configuration management in traditional SE could be facilitated by using a text-based version control system, the metadata, and data, about models and model transformations used in MDSE are more complex, and ordinary text-based version control systems do not suffice.

The second reason stems from the fact that models may be defined by using graphical notations, that are not yet standardized. In particular, for each metamodel a set of (different) graphical primitives is specified, and these primitives are used to create models *conforming* to that metamodel. This is analogous to the concrete syntax of programming languages in traditional SE. Therefore, in order to create models conforming to a specific metamodel a dedicated (graphical) editor for the models conforming to that metamodel must exist.

In the rest of this chapter, we first discuss basic ingredients of MDSE: models and metamodels in Section 1.1, and model transformations in Section 1.2. Thereafter, we discuss tools for developing models in Section 1.3, we give a short introduction to the field of software configuration management in Section 1.4, and we discuss model configuration management in Section 1.4.2. Next, since one of the main artifacts in model configuration management systems are model differences, in Section 1.5.1 we discuss the problem of model comparison and the problem of representing the difference between models. Moreover, since metamodels can also change during the design process, in Section 1.5.2 we discuss the process of adapting models in case their metamodels evolve.

Afterwards, in Section 1.6, we define research questions answered within this dissertation. Finally, in Section 1.7, we give an outline of the dissertation, and we relate each research question to chapters in which that particular question has been answered.

1.1 MDSE: Models, Metamodels and Metametamodels

As already mentioned, in MDSE models are described by using domain-specific modeling languages (DSMLs). This is similar to the traditional SE, where a program is described by using a programming language. A syntax of a DSML is described by a *metamodel*, and it is said that a model is an *instance of*, or that it *conforms to*, a metamodel. Thus, metamodels in MDSE play the role of (context free) programming language grammars in traditional SE. Furthermore, the syntax of a metamodel is described by using a metametamodel. Thus, the role of metametamodels is similar to the role that metalanguages (e.g. BNF, EBNF) play in traditional SE. The relations between models, metamodels and metametamodels, and the relations between these concepts and the concepts used in traditional SE, is depicted schematically in the Figure 1.1.

However, the actual, real-world, modeling frameworks are not fully consistent with the schema depicted in the Figure 1.1. we will elaborate on this subject in Chapter 2.

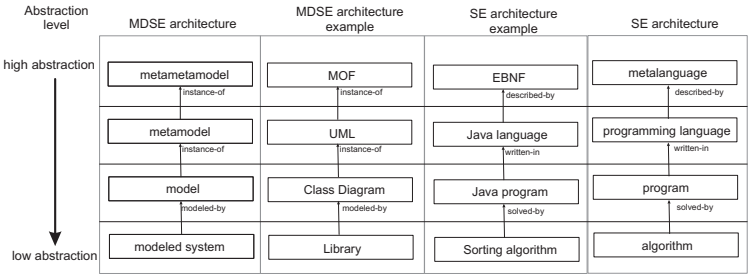

Figure 1.1: The relation between models, metamodels, and metametamodels and programs, programming languages, and metalanguages

In the next two sections, we will discuss two traditional, widely used, metametamodels, we will discuss the means of developing metamodels that are instances of those two metametamodels, and we will discuss the means of developing models that are instances of metamodels. The first metametamodel we discuss is a part of the MetaObject Facility framework, and is called *MOF 2 model* [20]. The second metametamodel we discuss is called Ecore [11]. Both metametamodels are self-describing, i.e. both metametamodels can be represented by models conforming to metamodels that are instances of those metametamodels.

1.1.1 MOF

The MetaObject Facility framework was constructed as an answer to the problem of how to specify a metamodeling framework to support Model-Driven Architecture (MDA) [19]. MDA is an approach to model-driven software engineering by the Object Management Group [24]. MDA proposes techniques and methods for the efficient implementation of MDSE compliant tools and frameworks. Note that MDA (and, with it, MetaObject Facility framework) is essentially just a set of guidelines, and does not include tool support. The MetaObject Facility framework follows the traditional metamodeling paradigm, as depicted in Figure 1.2[1]. In MOF, four levels of abstraction, labeled M0 to M3, are distinguished. Level M0 is the level of actual, real-world systems. Level M1 contains models, level M2 contains metamodels, and level M3 contains ”a language for building meta-

[1]Figure based on Figure 7.8 in [32].

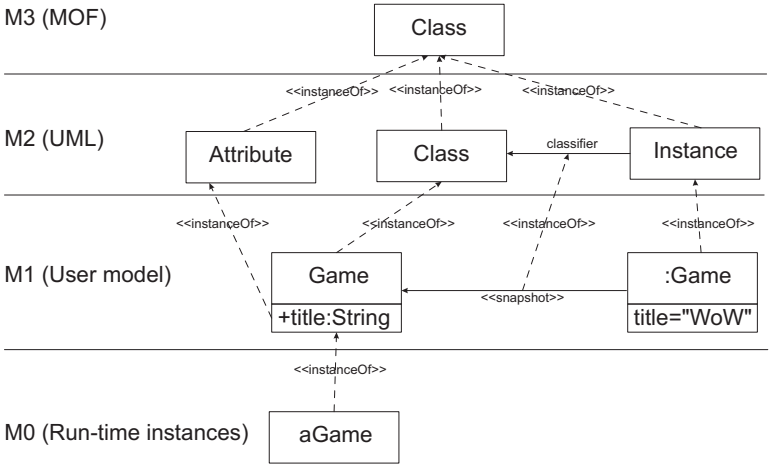

Figure 1.2: MetaObject Facility framework schematic (based on Figure 7.8 in [32])

models" or a metametamodel (this language is called the *M3-model* or a *MOF 2 model*). We will refer to the *MOF 2 model* as *MOF* in this dissertation.

MOF is based on the concepts found in object-oriented programming paradigm, and is described by using a subset of the Unified Modeling Language (UML) [98] graphical concepts. The three parts of MOF are *the Core*, *the Essential MOF* (EMOF) and *the Complete MOF* (CMOF). The architecture of MOF is depicted in Figure 1.3[2].

The Core consists of packages[3] that contain constructs that can be used to define metamodel elements. The two main metamodel element types in MOF are called *Class* and *Relationship*. An instance of a *Class* is used to model a class of entities. Each *Class* instance can have *Attributes* which are used to specify properties of a class of entities. Relations between classes of entities are modeled by connecting *Class* instances with *Relationship* instances. Metamodels that are instances of MOF are represented diagrammatically, by using the UML graphical notation for classes and relationships. In models that are instances of MOF metamodels, the instance of a *Class* instance is called *an Object*, and *Objects* are

[2]Figure adopted from `http://www.omg.org/spec/MOF/2.4/Beta2/PDF/`.

[3]A package is a container-type entity which allows grouping of related entities.

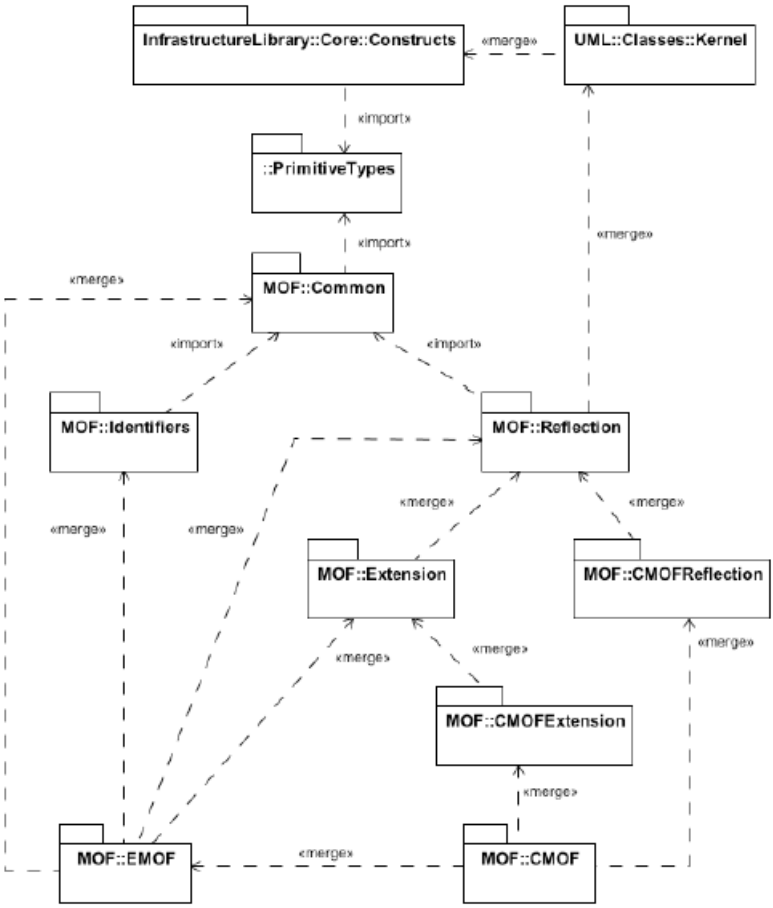

Figure 1.3: *MOF 2 model* architecture

connected by instances of *Relationship* instances.

EMOF contains extra packages for associating identifiers with metamodel elements, for extending the metamodel elements with new, unanticipated information, and for providing reflection capabilities to metamodel elements. CMOF

extends EMOF with further reflection capabilities, as well as extension capabilities, which are not important for this dissertation, and thus will not be discussed in detail [21].

1.1.2 Ecore

Ecore is a metametamodel which is based on the *KM3* [78] metametamodel. In essence, Ecore is a concise variant of MOF. Ecore is a part of the Java-based Eclipse framework [10], and uses Java in its description (in particular for describing data types), just like MOF uses parts of UML in its description. Furthermore, like MOF, Ecore is described by using a UML-like graphical notation. The main components of Ecore and the relations between them are depicted in Figure 1.4 (taken from [11]).

The main components of Ecore are, similarly to MOF, classes and relations (the latter are called references in Ecore), but also packages, operations and attributes. However, unlike MOF, which provides only a set of guidelines for the definition of modeling and metamodeling tools, Ecore is geared towards implementation. In particular, Ecore is implemented as a set of Java classes. This set of classes is incorporated in a framework which provides methods for creating and editing Ecore metamodels and models. The Ecore metamodels and models are thus represented as Java classes, but can be serialized, and persisted in files. Ecore metamodels are instances of Ecore, and Ecore models are instances of those metamodels.

Although the two discussed metametamodels differ in many aspects, both of them can be considered as equivalents of general purpose languages in the modeling world— i.e. general purpose metametamodels. This is the case because they both provide constructs (e.g. hierarchies, modules, inheritance, data types, etc) for dealing with all kinds of possible situations one might encounter while modeling.

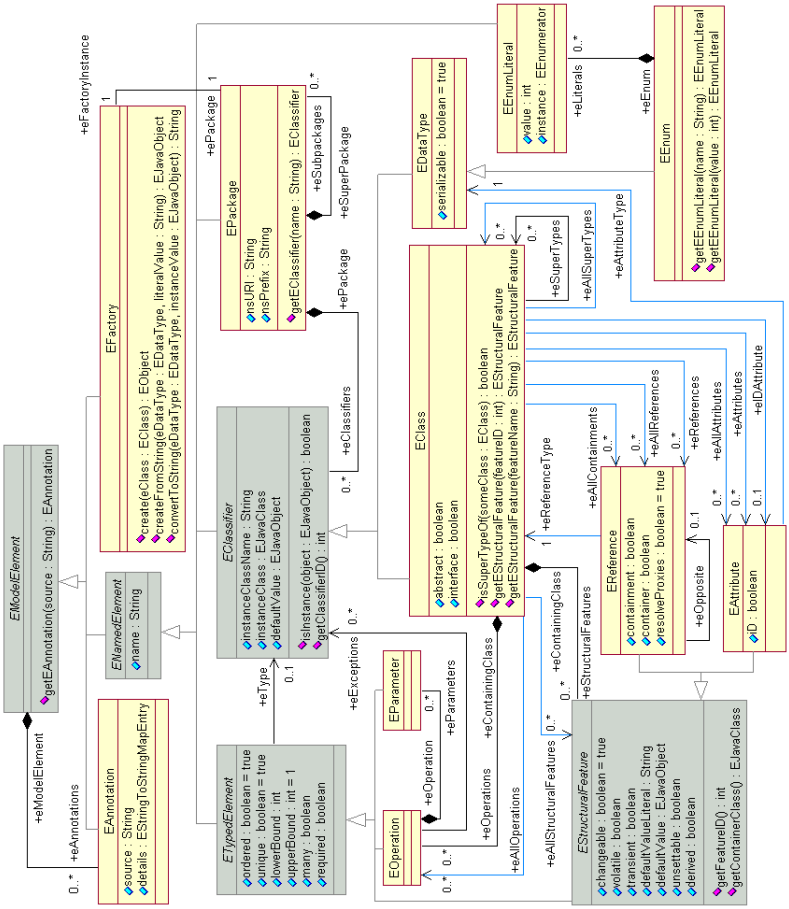

Figure 1.4: Ecore component architecture

1.2 MDSE: Model transformations

In MDSE, model transformations are used to transform one model (source model) into another (target model). The models may conform to the same, or different metamodels. The use of automatic transformations is recommended, but manual transformations are also possible.

There are many ways to classify model transformations [54, 94]. One possible classification distinguishes between horizontal and vertical model transformations [94]. Horizontal model transformations are used to transform a model into another model having the same metamodel. These transformations can be used, for example, for refactoring. Vertical model transformations are used to transform a model into another model having a different metamodel. These transformations have two main usages. The first usage is in refining a model (if the transformations transform a more abstract model into a more concrete model), or in abstracting a model (if the transformations transform a more concrete model into a more abstract model). Refining is used to improve the design of a model. Abstracting is used to provide better insight into the parts, or the relations between the parts, of a model. Another main usage of vertical model transformations is in transforming a design model into, for example, a verification, validation, or a simulation model. In this case, the model is transformed into another model that is semantically loosely related to the original model (while in cause of refinement or abstraction, both the initial and the transformed model are semantically strongly related). However, this other model can be used to check some properties of the original model which are hard to check in the original form. The main challenge in specifying a vertical model transformation is that the syntax and the semantics of metamodels of the source and the target model differ. This difference opens a syntactic and a semantic gap that needs to be closed by a model transformation. However, while it is relatively easy to overcome the syntactic gap, closing of the semantic gap is not an easy task as shown in [109].

In another classification, the categories of imperative or declarative transformations are distinguishable [94]. Imperative transformations are specified in some imperative language. An example of an imperative transformation would be a Java program for transforming Ecore models. Declarative transformations are specified by a set of declarative statements (rules). A transformation engine reads the defined rules and applies them to a source model, producing a target model. An example of a declarative transformation would be a transformation for transforming Ecore models specified in Atlas Transformation Language (ATL) [1], QVTr language [25], or VIATRA2 [33].

1.3 MDSE: Tool support

Unlike in traditional programming languages, where a text editor was sufficient to start working on a program, in MDSE specialized tools are essential. As already mentioned, one of the main reasons for this is that models are created by using dedicated graphical symbols, and thus cannot be edited by using a textual editor. Another reason is that in the MDSE paradigm models are the main design artifacts, and MDSE promotes model transformations as a preferred way of transforming models between steps of the development process. Thus, models and model transformations should be formally managed, which requires the existence of a model configuration management system.

In the rest of this section we will discuss three mature open-source modeling tools. Although there are many commercial modeling tools (Rational Rhapsody [26], Rational Rose [27], Enterprise Architect [12], etc.), we focused on open-source tools because they are free, and it is possible to get insight into the precise details of their design and functionality, which is important in order to discuss them at the appropriate level of details. However, there does not exist an open-source model configuration management system (MCMS). Thus, we will not discuss any MCMS. Nevertheless, in Section 1.4 we will describe the major requirements that any MCMS is expected to fulfill.

The three modeling tools that we discuss differ in scope and generality. The first tool is *ArgoUML*, which can be used to create and edit UML models. Thus, the scope of ArgoUML is quite limited, since it is based on one particular metamodel—UML. ArgoUML supports the creation of all nine types of diagrams defined in version 1.4 of the UML. Thus, the generality of ArgoUML is also quite limited. However this is to be expected, since ArgoUML supports only one metamodel. A screenshot of ArgoUML is depicted in Figure 1.5. ArgoUML is a perfect example of an elementary MDSE tool: it provides a hierarchical tree-like overview of the model in the left part of the main window, the center part of the main window is reserved for a diagram editor, and in the bottom part of the main window the properties of the model element selected in the edited diagram can be inspected and changed.

The second tool that we discuss is a Java-based *Eclipse* framework [10], and it

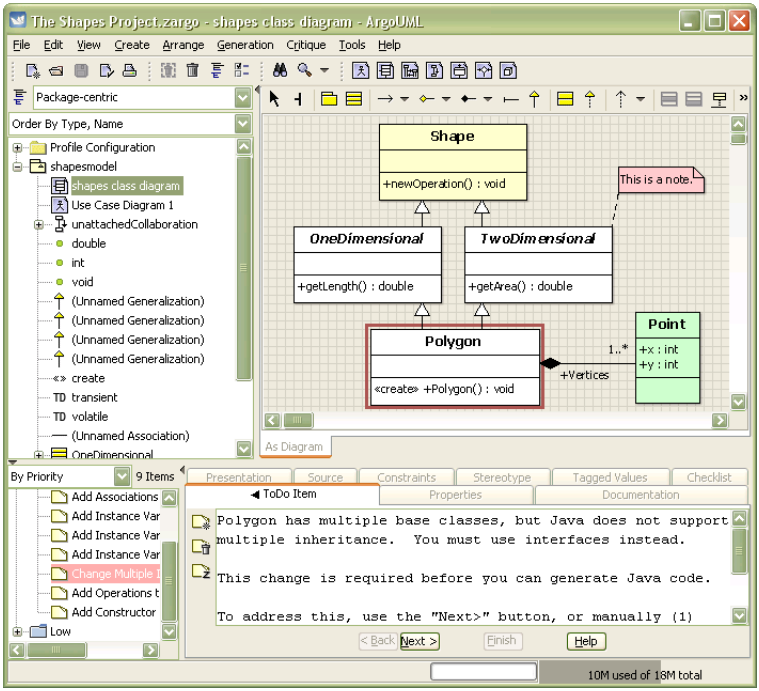

Figure 1.5: An ArgoUML screenshot

can be used to create and edit Ecore-based metamodels and models. Eclipse has a much larger scope than ArgoUML, since it provides not only model development facilities, but also metamodel development facilities. Actually the scope of Eclipse is limited only by the capabilities of the Ecore metametamodel. Eclipse is also more generic than ArgoUML, which is also to be expected since Eclipse is not tied to a specific metamodel. But, this generality comes with a price: the concrete syntax (i.e. graphical primitives) must be defined for each metamodel by using a Graphical Editing Framework (GEF). This is also something to expect since the models for warehouses, petri nets, or class diagrams use different graphical primitives. However, efficient use of the GEF requires extensive knowledge of the Java programming language, as well as extensive knowledge of the internal functioning of the Eclipse framework itself, which limits the generality of Eclipse.

The third tool that we discuss is the Generic Modeling Environment (GME) [17], which is truly generic. The scope of GME is the same as Eclipse—GME provides facilities for developing metamodels as well as models. However, unlike Eclipse, in order to use GME (which is also created by using the Java programming language), metamodel designers do not have to know Java. As in Eclipse, the concrete syntax of models must be defined, but it is defined by using a special *aspect* architecture provided by GME. The aspects can be interpreted as different types of diagrams for models conforming to a specific metamodel. However, although much more concise, and easier to use than the GEF, the definition of the concrete syntax of models is still a tedious task. Thus, it is my opinion that both Eclipse and GME, or any other future generic modeling environment, require a full-fledged visual meta-language.

In the next section we discuss an important aspect of engineering in general, namely the process of *configuration management*.

1.4 Configuration management

Configuration management (CM) is an important part of any production process [44, 53, 81, 86]. CM includes identifying, capturing, organizing and disseminating all important constituents of the production process. For example, CM in a car factory would include capturing, organizing, and disseminating information on car design documents, car parts availability, car production status, etc.

In this dissertation, we focus on software configuration management (SCM). SCM is a subfield of configuration management, specialized for the software production process. In particular, SCM deals with identifying, capturing, organizing, and disseminating files constituting a software project. SCM (also called *Revision Control*) includes some activities used in general CM, like configuration identification, auditing, status accounting, and control, but it also introduces several new activities. In order to discuss these activities in detail, the concept of a revision must be explained. However, before defining a revision, it must be mentioned that in all SCMs it is possible to manage multiple *versions* of a software artifact (i.e. a file). Revision is defined as ”a new version of an item that

is intended to replace the old version of the item" [29]. Each revision of an item is assigned a unique identifier (revision identifier). A closely related term to the term revision is a term *variant*. However, while *revisions* are used to describe a chronological evolution of an artifact, the term *variant* describes the parallel evolution of the same artifact.

The basic SCM activities are [29]:

- *Configuration identification* in the context of a SCM concerns identifying and gathering the correct set of artifacts for a certain version of a software product. The artifacts put under version control are called configuration items in the context of a SCM system. A *revision* of a software product is an identifier assigned to a software product, that enables to track the evolution of a software product. For example, a software product identified with revision 1 was released before the (same) software product identified with revision 2.
- *Configuration status accounting* concerns recording and reporting the *baseline* of a configuration item in a configuration. As defined in [29]: "A (software) baseline is a set of software items formally designated and fixed at a specific time during the software life cycle".
- *Configuration auditing* concerns checking that the functional and performance requirements of an entire configuration, or of a specific configuration item, are satisfied.
- *Configuration control* concerns with a set of rules and guidelines for approving the change to a configuration item in the *baseline*.

Due to the specific properties of software (software exists only in a digital form), SCM introduces some specific activities like release management and defect tracking. *Release management* is related to gathering and organizing the information on build environments, tools, and scripts, which are required for producing a specific release of a software system. Moreover, it concerns setting the criteria for deciding when the status of a versioned software system may change (e.g., a status may change from *in_development* to *released*). *Defect tracking* is related to tracking defects in the software. This activity allows the developers to link a software defect to a certain configuration item (or items), such that the

defect can be eliminated in a future version of the software.

It is important to mention that a successful application of SCM to a software project depends on the SCM tools. However, the complexity of an employed tool should be correlated with the complexity of a project and the size of a company—setting up a SCM and requiring a strict adherence to SCM rules in an overly complex tool creates a (unneeded) burden for small projects or small teams.

1.4.1 Repositories and versions

In a project which is not managed by a SCM system, the project files are stored in a long term memory, such as hard disk. In a SCM managed project, these files are stored at special locations called *repositories*. Repositories are also long term memory, usually stored on hard disks, and the structure of repositories reflects the structure of a file system. However, repositories have special properties related to storing and retrieving files. In particular, repositories are capable of storing multiple versions of the same file, and once stored, permanent deletion of the file is not allowed.

Based on the type of a repository, there are two types of SCM systems. One type are the client-server systems, having a central repository. This repository is located on a computer designated as a server, and clients access this repository to obtain and update stored configuration items. Example systems of this type are SVN [28] and CVS [6]. Another type of SCM systems are the distributed systems, having a distributed repository. Thus, in distributed SCM systems, each client maintains its local repository, and merging algorithms are used to keep all local repositories synchronized. Example systems of this type are Mercurial [22] and GIT [16].

Since each SCM system introduces its own terminology, and introduces terms not used by other SCM systems, we will explain the concept of versions by using the terminology specified by a client-server SCM system called SVN [28]. In the terminology of SVN, the configuration items are files residing in a hierarchical (directory-like) structure. Each file has an associated revision identifier. The server repository also contains all older versions of each file, and it is possible

to access those versions. Sets of versioned files are organized into *branches*. There is always one *main branch*, which is accessed by default by clients. Files in the main branch can be *tagged* to create a logical group. Tags are a common mechanism to denote different releases of a software product.

The set of (latest versions of) all files in the main branch is called *a baseline*. Notice that the baseline revision identifiers can differ for different files. The copies of all files from a particular server baseline at a client will be called a *client workspace*. Clients obtain (copies of) files from a server by invoking initially the *CHECKOUT* operation, and subsequently the *UPDATE* operation.

Changing one or more files in the branch (also called *patching*) assigns new revision identifiers to those files. All files that have been changed together in one *patch*, receive the same revision identifier. The patching is initiated by clients, by invoking the *COMMIT* operation. A successful *COMMIT* operation transfers the contents of selected files in the client workspace to the repository, adding new versions of the changed files on server.

The graphical representation of an example versioning process is depicted in Figure 1.6. In the example versioning process, initially two files *a.txt* and *b.txt* are put under version control. Later, file *a.txt* is changed, and gets a new revision identifier. The new configuration is tagged as *V1.0*. Next, a new branch is created, and a new file *c.txt* is added to the new branch. Thereafter, the new branch is *merged* into the main branch, inserting the file *c.txt* into the main branch. The final configuration is tagged as *V2.0*, and it is a new *baseline*.

One of the requirements for a SCM system is the efficient storage of files. This is achieved by not storing all the versions of the evolved file, but by storing only the initial version, and the differences between subsequent versions (sometimes the latest version and the difference between the latest and the previous versions are stored). This works because the combined size of an old version of a file and the difference between old and the new version usually is smaller that the combined size of an old and a new version of that file (though with really small files, or with packed files, it might be the case that combined size of the old file and the differences is larger than the combined size of the old and the new file).

There are two main approaches for obtaining the difference between two files. In

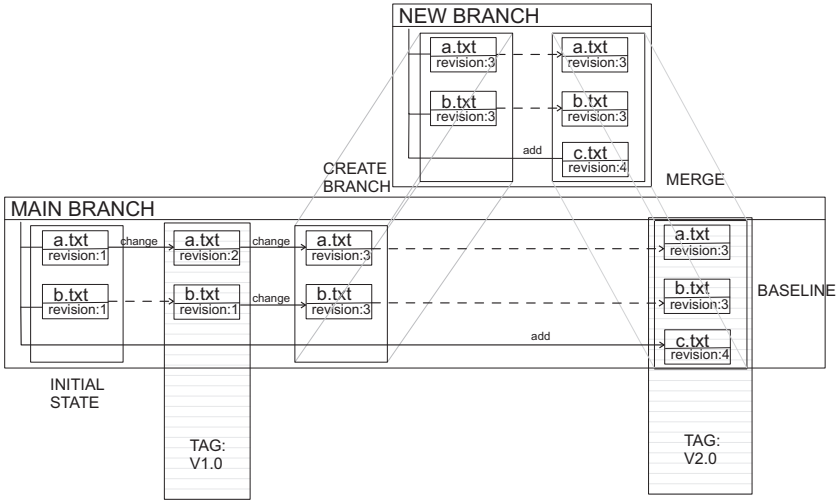

Figure 1.6: Example versioning process

the first approach, which is called *state-based*, the difference is calculated by a special *differences calculation algorithm*. This calculation algorithm receives as arguments an initial and a target file, and returns their difference. The returned difference is a set of atomic differences that can be used together with an old file as arguments of an inverse calculation algorithm to obtain the new file (i.e. this set can be used as a patch). In the second approach, which is called *operation-based*, the difference consists of a set of operations supplied by editing tools. This set of operations can be used to transform an initial file into a target file.

As an example consider the difference between numbers 63 and 65. A state-based differences calculation algorithm would return the difference as, for example, $2 \rightarrow 5$, noting that the second character of the first number should be replaced by a character 5. An operation-based difference would be, for example, $+2$, denoting that, in order to obtain the second number, one should add 2 to the first number.

In the example versioning process, the concept of *merging* branches was mentioned. This concept is very important and warrants more explanation. Because the concept of *merging* is intertwined with concepts of *optimistic* and *pessimistic*

approaches to version control, we will discuss these concepts in detail in the next paragraph.

Optimistic and pessimistic version control

The optimistic and pessimistic approach to version control are related to the possibility of parallel development of a software system. In the pessimistic approach to version control, the baseline is *locked* by one designer. This means that only the designer that has *locked* the baseline can change the baseline, and no one else is permitted to change it until the designer that holds the lock releases the lock. Thus, in the pessimistic approach, only one developer at a time can effectively work on the system. In the optimistic approach the baseline is not locked, and anyone can change it at any time. Thus, in this case, multiple developers can work on the system at the same time.

In both approaches, there is a possible problem when combining the contents of the *main branch* and the contents of a *client workspace*. This problem, referred to as the *merge* problem, occurs when two clients change the same versioned artifact, in a different way. An illustration of this problem is depicted in Figure 1.7.

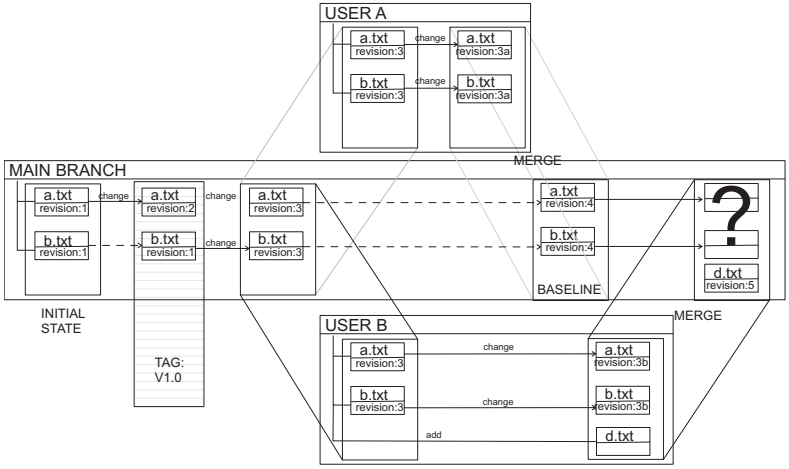

Figure 1.7: Example merging process

In the example process, designers *A* and *B* initially *UPDATE* all files from the main branch to their local workspaces. Next, designer *A* changes files *a.txt* and *b.txt* and *COMMITS* the contents of his workspace. In this case, committing consists of replacing the configuration items in the main branch with the related configuration items from the client workspace. The differences between configuration items in this case are called *"2-way" differences*. However, if the designer *B* changes the same files with different changes, and tries to commit his changes after the designer *A* has committed his files, then the changes that the designer *A* has made will be overwritten unless they are exactly the same as the changes made by designer *B*. This problem must be solved by using *merge algorithms*, to ensure that the changes introduced by both designers *A* and *B* are consistently incorporated in the final baseline configuration item. The differences between configuration items in this case are called *"3-way" differences*.

Notice that the problem of merging exists in both the pessimistic and optimistic approach to version control. The only difference is that in pessimistic approach to version control the designer who has the *lock* can safely copy the contents of his local workspace to the main branch, but all the other designers that are working on the same model must employ merging algorithms (after they obtain the repository lock).

1.4.2 Model configuration management

In this section, we will discuss model configuration management (MCM), and the requirements that MCM systems should fulfill.

MCM is a specialization of SCM, with models as the configuration items. The differences between MCM and SCM stem from the fact that MCM should be used in model driven software engineering, where the formal requirements on versioned artifacts are much stricter than in SCM. For example, all versioned models must have an associated metamodel, which is not the case with all versioned text files. Furthermore, metamodels change during the development process, thus metamodels should also be versioned, and included in the configurations. This can be considered as an extension of the release management activity in SCM, and will be called *metamodel management*. Also, since models change

by using formal model transformations, these model transformations must also be managed by a MCM system. The part of the MCM that manages model transformations will be called *model transformation management*. Next, since models, and model transformations, greatly depend on tools, tool-specific information should also be managed. This part of the MCM will be called *tool management*.

Finally, it is important to mention that models need to be persisted in order to be versioned. However, different tools use different mechanisms for persisting models. This creates a problem of managing models persisted by different tools. This problem does not exist in MCM systems which are based on operation-based differences, since these systems have a predefined format of differences that they expect from tools. However, in MCM systems utilizing state-based differences, this problem is very important since the differences calculation mechanisms must be adapted to support all persistence mechanisms. One possible solution to this problem is to define a common metametamodel, and to represent all models, metamodels and model transformations by using this metametamodel. This allows MCM systems to use the same differences calculation and representation mechanisms with all models. However, this solution also requires the definition of bi-directional transformations of models supplied by each tool, to a common representation. In this dissertation, we follow this approach.

1.5 Model differences and model co-evolution

As already mentioned in Section 1.4.1, in order to have an efficient MCM system, only the differences between two successive versions of a model should be stored, and shared with clients of the MCM system. For this purpose methods for *representing*, *calculating*, and *processing* model differences should be developed and used [84]. Note that this holds in both state-based and operation-based approaches to versioning. In this dissertation, we focus on the state-based approach to versioning (and discuss state-based model differences). However, in Section 1.5.1 we will also briefly discuss the operation-based model differences.

Furthermore, in case that a metamodel evolves, it is required to adapt all models in a MCM system, that conform to the old version of the metamodel, in order for

them to conform to the new version of the metamodel. This process is known as metamodel and model co-evolution (though we will refer to it as model co-evolution), and we will discuss this process in Section 1.5.2.

1.5.1 Model differences

In this section, we will first discuss a set of requirements that model differences should fulfill in order to be used in the context of Model Driven Software Engineering. This set of requirements has been introduced by Cicchetti in [49]. Next, we will discuss the three main aspects of model differences: representation, calculation, and processing (e.g. visualization).

Requirements

In order to use the model differences in the context of a Model Driven Software Engineering, they should satisfy the following set of requirements:

- Model based: The differences should be represented by a formal differences model (i.e., a model conforming to a differences metamodel).
- Transformative: It should be possible to transform one model into another model using a differences model (i.e., it should be possible to use model differences as a patch).
- Self-contained: The differences model must contain all the information autonomously without relying on data contained in the compared models.
- Minimality: The differences should contain a minimal number of entities.
- Metamodel independent: The differences metamodel should be independent of a particular metamodel (e.g. UML).
- Layout independent: The differences metamodel must be agnostic of presentation issues.
- Invertible: It should be possible to revert back to the old model using the new model and their differences model.
- Compositional: The result of the sequential or parallel modifications is a differences model whose definition depends only on difference models being composed and is compatible with the induced transformations.

Since in MDSE environments everything is either a model, or a model transformation, the differences should be model based and transformative. The self-contained requirement and the minimality requirement are related, because they capture the idea that the differences model should contain all the differences and only the differences. The differences should be metamodel and layout independent because they should be usable in generic environments to allow the building of domain specific comparison frameworks. The differences model should be invertible in order to allow the users of a MCM system to easily obtain an old version of a model from a new version of the model. The compositional requirement will be discussed in more detail in the following paragraph. The specified requirements are taken from the work of Cicchetti [49], and a more detailed discussion on these requirements can be found there.

Representation

As mentioned in previous section, in order for model differences to be used seamlessly in MDSE, the difference between two models should be represented by a *difference model*. Furthermore, difference models, as all other models, should conform to a difference metamodel. The difference metamodel should allow the description of the difference between models in both common usage scenarios, i.e. the differences metamodel should be able to describe both "2-way" differences and "3-way" differences. This is related to the compositional requirement for model differences: the "2-way" model differences can be considered as a result of a sequential modification of a model, and the "3-way" model differences can be considered as a result of a parallel modification of a model. However, the differences metamodels for state-based model differences and operation-based model differences are bound to differ. This is because in state-based approaches, the differences are the result of a differences calculation algorithm, and are essentially just *data*, while in operation-based approaches the differences must represent both the *operations*, and the *data*.

An example metamodel for the description of state-based model differences between *UML* based models, as specified in [50], is depicted in Figure 1.8.

Notice that for each UML element (i.e. Class, Attribute, Parameter or Operation) three additional elements are defined. The instances of these additional

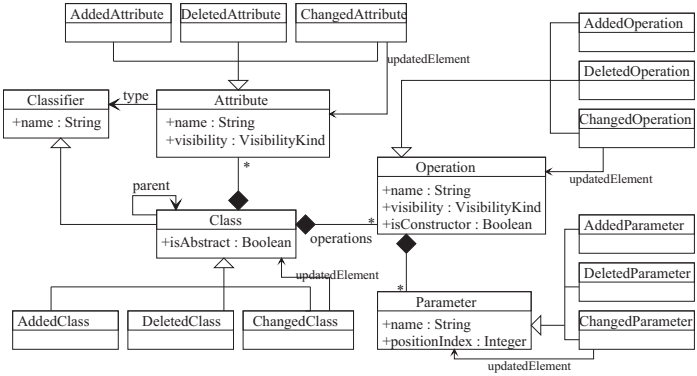

Figure 1.8: State-based model differences metamodel for UML models

elements model added, deleted, or changed element instances between two compared models.

An example metamodel for the description of operation-based model differences between *Ecore* based models, defined in [70], is depicted in Figure 1.9.

Notice that unlike in the state-based differences metamodel, the focus of the operation-based metamodel are model operations. In particular there are create and delete operations, which model adding or deleting complete model elements, and there are feature operations, which model the change to model elements (because in Ecore features represent attributes of model elements, or relations between model elements).

Both described approaches have deficiencies, which will be discussed in Chapter 3. Moreover, in Chapter 3, we describe our approach to the representation of state-based model differences, that solves some deficiencies of the existing representations of state-based model differences.

Calculation

The calculation of model differences is used in both the state-based and operation-based approaches to model versioning. However, since in this dissertation we focus on state-based approach to model versioning, we will describe a calculation

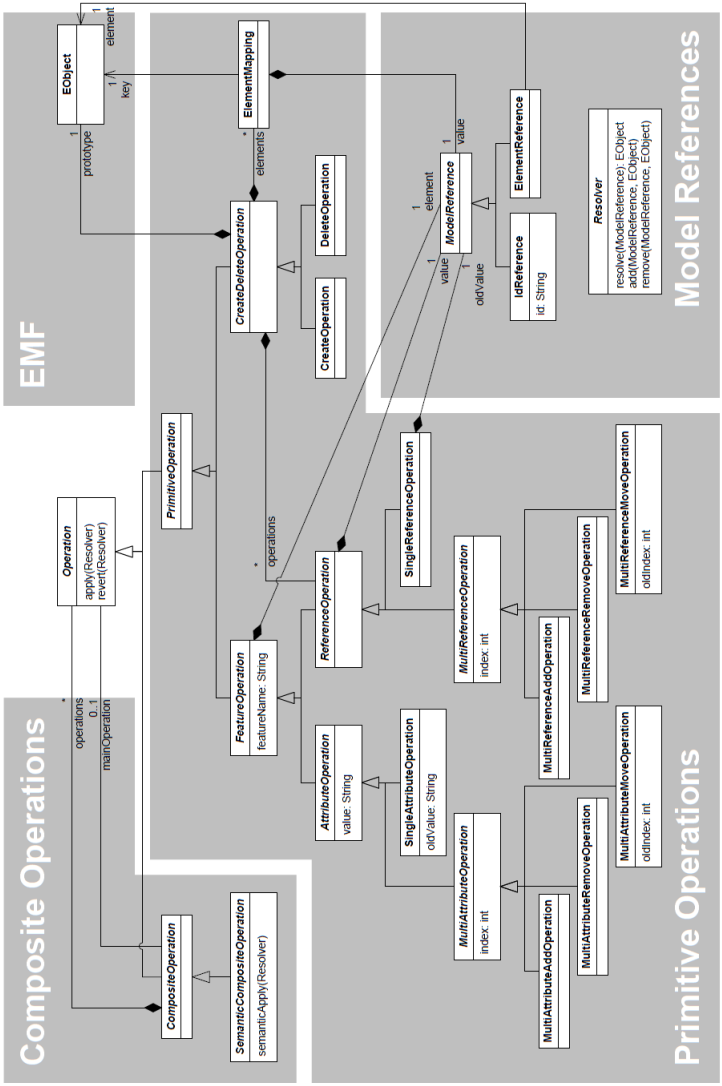

Figure 1.9: Change-based model differences metamodel for Ecore models

algorithm for that case.

As already specified in the requirements (in particular, the minimality requirement), the goal of model differences calculation algorithms is to produce differences models having a minimal number of elements. Although the difference representation mechanisms should be applicable in case of both "2-way" and "3-way" differences, the calculation algorithms differ greatly in these two scenarios. The difference between these concepts is so big, and the research involved is so broad, that this dissertation focuses predominantly on "2-way" differences. However, at the end of this section we will give a brief overview of algorithms for calculating "3-way" differences.

"2-way" differences calculation algorithms usually consist of two phases. In the first phase a *matching* of two compared models is done. In the second phase, based on the found matching, the differences are calculated. The matching of the two models is a mapping between elements in one model and elements that (may) represent the same entity in another model. The minimality requirement in a "2-way" differences calculation algorithms is achieved by calculating a *maximum matching* between models being compared. The maximum matching is the matching that matches the maximum number of elements. There are four distinguished matching strategies [84]: static-identity, signature-based, similarity-based and language-specific, which are discussed below.

Static-identity based matching assumes the existence of universally unique identifiers (UUID) that are assigned to model elements upon creation and that are persisted together with model elements. Since each entity in the modeled system should be represented by only one model element, in order to have a consistent model, in this approach model elements that have identical UUIDs are matched. This approach is most applicable in case of sequential model development, where only one designer works on a model during a certain period of time. The reason for this is that if two users would work on the same model at the same time, it could happen that they both model the same entity, but by using model elements having different UUIDs. Moreover, if a user accidentally deletes a model element, and re-creates a model element representing the same entity, the matching algorithm could not match these two elements, although they represent the same entity.

Signature-based matching assumes that for each model element, a uniquely identifying *signature* can be calculated based on features of the model element. The signature can be, for example, a string obtained by concatenating the names of all ancestor elements, and the name of the selected element. The elements that have the same signature are matched.

Similarity-based matching requires a similarity function that calculates the similarities between two model elements. The similarity function usually returns a normalized value (i.e. between 0 and 1), and elements with similarity value greater than a certain threshold value are matched (e.g. all elements with similarity greater than 0.5).

Language-specific matching assumes a matching algorithm particulary tailored to a specific modeling language (i.e. these matching types are usually metamodel-dependent). Thus, matching algorithms of this type usually use both syntactic and semantic information to achieve as best as possible matching for a particular language. There are two types of approaches in defining the language-specific matching algorithms. In the approaches of the first type, the matching algorithm is defined as a set of matching rules. Thus, this is a declarative approach to matching. An example of this approach is the Epsilon Comparison Language [13]. In the approaches of the second type, the matching algorithm is defined through a program in an imperative language. An example of this approach is EMFCompare [9], where the metamodel-independent matching algorithm is defined in Java.

All four mentioned matching strategies perform well, if a particular set of conditions is fulfilled. However, neither of the four is a *silver bullet* [45] that provides the best matching in all cases.

Based on the matching found, the differences are calculated in the following manner: Assume that the input to the calculation algorithm are models A and B. Next, assume that the matched elements are given as a set $M(A,B)$. Then the difference model would contain a set of deleted elements $A - M(A,B)$, a set of added elements $B - M(A,B)$, and the set of changed elements $changes(M(A,B))$ (the function *changes* calculates the changes between matched elements). If the two matched elements are completely identical then the change is empty, and

otherwise the change contains the differences between the contents of model elements.

"3-way" differences calculation algorithms are more complex and, unlike "2-way" differences calculation algorithms, actively include the human operator in the calculation process [4]. There are two main sources of complexity of "3-way" differences calculation algorithms. The first source of complexity is the fact that there are three models involved: the client baseline model (A'), the model in a client workspace (A), and the server baseline model (B). Notice that the client baseline model A' is also the ancestor version of a server baseline model B. The second source of complexity is the fact that the compared models A and B can be in *conflict*. A conflict occurs if two designers change the same model element in a different way. For example, one designer may change the title of a class *User* to *Client*, and another designer may, in parallel, change the title of the same class to *Customer*. Resolving these conflicts is not trivial, and has been addressed in numerous works, e.g. [47, 82, 121]. An implementation of a "3-way" state-based model differences algorithm, can be found in EMFCompare [9].

Processing

After the differences have been calculated, they can be processed. The most common use of model differences is a *patch*.

Another possible usage of model differences is in exploring the evolution of models. Since models are represented diagrammatically, this includes the visualization of model differences. Usually, the model differences are visualized by using a *unified view* on differences. In this approach, the differences are superimposed on the old version of the model, and colors are used to highlight the meaning of differences (e.g. green color is used for added elements, red color is used for deleted elements, and blue color is used for changed elements). Another possibility is to use a separate view approach, where both models are visualized in parallel, and the differences are also highlighted by using colors.

[4] It is possible to prove that it does not suffice to use only the combination of "2-way" differences between models A and A' and between models B and A', to obtain "3-way" differences, but the proof is out of the scope of this dissertation.

1.5.2 Model co-evolution

It is often the case that metamodels evolve during the design process or during the maintenance of a software system. In those cases, it is often required to adapt the models conforming to the initial metamodel, such that they conform to the evolved metamodel (otherwise they can be marked as legacy models). This process is denoted as *model co-evolution* (or *coupled evolution of metamodels and models*).

In order to perform the co-evolution of models, two sub-problems need to be solved. The first problem is how to calculate the differences between the initial metamodel and the evolved metamodel. The second problem is how to adapt the models based on the calculated differences. Existing approaches for solving the co-evolution problem greatly differ depending on whether differences between models are known (operation-based approaches) or are not known beforehand (state-based approaches).

In case the differences between two metamodels are known beforehand, the first subproblem of model co-evolution disappears. An example approach that assumes this is COPE [71]. Moreover, in COPE it is assumed that the differences between metamodels are operation-based. An extensive list of metamodel operations, supported by COPE, is described in [72]. The second subproblem in COPE is solved by splitting a set of all possible metamodel operations in two subsets[5]. In the first sub-set are the operations for which the syntactic and semantic influence on co-evolving models is known. For these operations it is possible to automate the co-evolution process completely. An example of this kind of operation is an operation that changes the name of a metamodel element. In this case, the co-evolution is trivial, since the change of the name of a metamodel element does not have influence on models. In the second subset are the operations for which the syntactic and semantic influence on co-evolving models is not known. For these operations a manual intervention is needed to co-evolve the models. There are two possible types of manual interventions. In the first type, the user that performs the co-evolution can create an *evolution script* (based on an operation), and this script can be used to automatically co-evolve models with

[5] In COPE it is assumed that the metamodels are instances of Ecore, and thus all possible (atomic) operations on metamodels are known.

respect to this operation. An example operation of this type is changing the type of an attribute of a metamodel element. In this case, a user must define a transformation function that can be used to transform values of the initial type to the values of the evolved type. In the second type, it is not possible to automate the co-evolution process, and each model must be manually co-evolved with respect to the selected operation. An example operation of this type is adding a reference between metamodel elements. In this case, a user needs to connect all the model instances that should be connected by the instances of this reference.

In case the differences between two metamodels are not known beforehand (i.e. the differences are state-based), the first subproblem needs to be solved as well. An example approach that assumes this is an developed by Garcés et. al. [63]. In their approach, a heuristic metamodel comparison algorithm is used to compare two metamodels. The differences between two metamodels are described by using a differences metamodel. Based on the differences found, the co-evolution is performed by first executing a higher-order transformation, that generates a transformation that can be used to evolve all models conforming to the initial metamodel.

Both presented approaches have advantages and disadvantages. A positive side of COPE is that, since the differences between evolved and initial metamodel are known beforehand, it is possible to do a more precise co-evolution. However, COPE requires that the differences between evolved and the original metamodel are known beforehand, which is not easily accomplishable. A positive side of the approach by Garcés et. al. is that it is tool-independent (though still metametamodel dependent). However, by using a custom matching algorithm for matching metamodels, the quality of calculated metamodel differences may not be that high, and the co-evolution could be imprecise. Moreover, it is much more complex to adapt the higher-order transformations, or even generated transformations, when a completely automated co-evolution is not possible.

1.6 Problem statement

Based on the previous discussion, the following summary can be composed:

> Model Driven Software Engineering is a paradigm that is quickly replacing traditional Software Engineering as means of developing, managing and maintaining complex software systems. However, MDSE depends on tools more than traditional SE. While the tools for developing models are maturing quickly, the tools for managing models are far behind in development. In this respect, better methods and tools for comparing models, for visualizing the resulting differences, and for co-evolving models, are needed.

Based on the above summary, we have formulated three main research questions. In this dissertation we discuss, and answer, these research questions.

Research Question 1. *How can the quality of methods and tools for model comparison be improved?*

Research Question 2. *How can the quality of methods and tools for visualization of model differences be improved?*

Research Question 3. *How can the quality of methods and tools for co-evolving models be improved?*

The first main research question can be further split into these subquestions:

- Which are the existing methods and tools for comparing models?
- What are the aspects of existing methods and tools for comparing models that can be improved?
- How to improve existing methods and tools for comparing models, and how to measure these improvements?

The second main research question can be further split into these subquestions:

- Which are the existing methods and tools for visualization of model differences?
- What are the aspects of existing approaches to visualization of model differences that can be improved?

- How to improve existing methods and tools for visualization of model differences, and how to measure these improvements?

The third main research question can be further split into these subquestions:

- Which are the existing methods and tools for model co-evolution?
- What are the aspects of existing approaches to model co-evolution that can be improved?
- How to improve existing methods and tools for adapting models in case their metamodels evolve, and how to measure these improvements?

In the next section, we give the outline of the rest of dissertation, and we relate each research question to the chapter (or chapters) where it is answered.

1.7 Dissertation outline

This dissertation summarizes our research on model evolution and model co-evolution, and the structure of the dissertation (except Chapter 2) reflects the evolution of that research.

In particular, in Chapter 3 we discuss the problem of model comparison, and we propose a new method for comparing models. The proposed method is based on, and improves, several existing methods for model comparison, therefore we also discuss these improvements. Chapter 3 answers Research Question 1 and is based on the paper that was presented and published at IWMCP 2010 [111]. However, because Chapter 3 is based on a workshop paper, where certain details of our approach related to model matching could not be described because of the page limit, these details are described in Chapter 2. Moreover, some of these details are also discussed in a paper that was submitted to the special issue of SoSyM journal on models and evolution [110].

In Chapter 4, we discuss a method for assessing the quality of model comparison tools, we define a data set that can be used for this purpose, and we use the defined method, with the defined data set, to assess the quality of two model

comparison tools. One of the assessed tools is RCVDiff and the other one is EMFCompare. RCVDiff was developed by us, based on the research presented in Chapter 3. More information on RCVDiff can be found in a paper published at ME 2010 [113]. EMFCompare is a tool used in Eclipse framework for comparison of Ecore-based models. The defined method, data set, and the assessment results can be used as a benchmark data for future modal comparison tools. Chapter 4 provides a mean of experimental measurement of the improvements to methods for model comparison discussed in Chapter 3, and thus also partially answers Research Question 1. Chapter 4 is based on the paper that was presented and published at IWMCP 2011 [114].

In Chapter 5, we discuss the problem of visualizing model differences. We argue that the same methods and techniques can be applied for model differences as for other large-scale information content. Thus, we combine an existing technique for visualization of model differences, with a technique for visualization of large-scale information content. The combination provides better insight in the meaning of model differences than both combined techniques used separately. This chapter answers Research Question 2 and is based on the paper that was presented and published at IWMCP 2010 [112].

In Chapter 6, we discuss the problem of model co-evolution, and we propose a new method for co-evolving models. The specified method proposes a solution that is conceptually similar to solutions offered by other approaches in this field, but that brings several concrete improvements. For instance, we propose to use a specially defined metamodel to represent metamodels as models. This allows us to use techniques for model comparison, presented in Chapter 3, to compare metamodels. Furthermore, the resulting differences are used as an argument of the automatic co-evolution transformation. This chapter answers Research Question 3 and is based on the paper that was presented and published at TOOLS 2011 [115].

Finally, in Chapter 7, we given a summary of our answers to the formulated research questions and conclude the dissertation.

Chapter 2

An Alternative Modeling Framework

In this chapter, we will discuss technical details of the two main contributions of this dissertation. Both of these contributions are briefly described in Chapter 3, but both warrant a more detailed explanation.

The first main contribution is a modeling framework we developed. The reason for developing a new modeling framework is that traditional modeling frameworks under-specify the instantiation relations between different modeling levels. This *instantiation problem* will be discussed in detail in Section 2.1. The modeling framework we developed clearly specifies the instantiation relation between different modeling levels. The developed framework introduces a new metametamodel that uses the same concepts that are used in traditional metametamodels (e.g. classes, relations, inheritance, ...), but defines and connects those concepts in a different manner. Details of the framework are discussed in Section 2.2.

The second main contribution is a *differences metamodel* we developed, that leverages the new modeling framework, and can be used for the representation of models of difference between two models. The differences metamodel and the new modeling framework were designed in parallel, such that the differences between the models satisfy all the model differences requirements described in Section 1.5.1. The developed differences metamodel is described in detail in Section 2.3.

2.1 An instantiation problem in traditional meta-metamodels

The problem discussed in this section is visible in two of the most commonly used metametamodels - Ecore and MOF. However, this problem, if not accounted for, will be present in all layered modeling frameworks. The main cause of the problem is under-specification of relations between metametamodel elements, metamodel elements and model elements.

In both MOF and Ecore, the metametamodels are the focus and they are well defined. Under an *object-oriented* interpretation of the `<<instance-of>>` relation between metametamodels and metamodels, it is possible to specify a metamodel by *instantiating* a metametamodel. However, for models to be instantiated from metamodels, one cannot use the same `<<instance-of>>` relation as the one between metametamodel and metamodels. The reason for this is that the entities that appear in a metametamodel have a different syntax and semantics from entities that appear in metamodels. For example, if entities in metametamodels are considered *classes*, then the corresponding type of entities in metamodels are *instantiated classes* (or *objects*). However, these *objects* cannot be used as *classes*, since their syntax and semantics differ from the syntax and semantics of classes. Rather, if the same `<<instance-of>>` relation that was used to instantiate metamodels from a metametamodel is to be used to instantiate models from metamodels, the *objects* (in metamodels) must be *transformed* into *classes*.

An example of this problem is depicted in Figure 2.1, which is divided into four parts. The first (topmost) part contains a fragment of the Ecore metametamodel

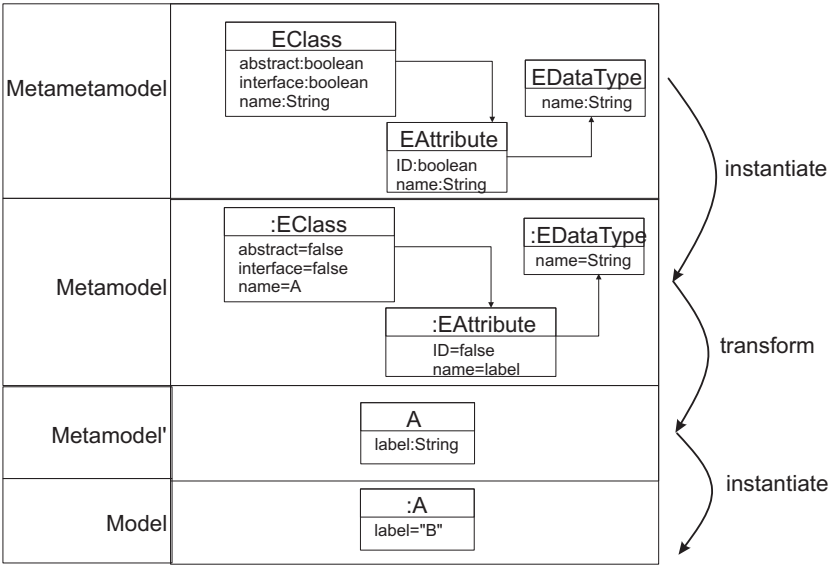

Figure 2.1: An example of the instantiation problem in layered modeling frameworks

that can be used to define metamodel elements. The second part contains a metamodel that is an instance of the Ecore fragment given in the top part. If entities in Ecore are considered *classes*, the elements of this metamodel are *objects*. Thus, this metamodel cannot be used to instantiate models. The metamodel' in the third part is obtained by transforming the metamodel depicted in the second part, by a transformation that transforms *objects* into *classes*. The metamodel in the third part can be used to define models, which was exemplified in the fourth part.

The transformation from objects into classes is not formally defined, neither in MOF nor in Ecore. Consequently, the notion of models is specified incompletely in traditional metametamodels. Though, e.g. in case of Ecore, this notion can be understood by examining the source code of Java classes that implement Ecore. This problem is particulary detrimental in case of calculating differences between models, since the calculation algorithms must take into account details of the concrete syntax of models. However, if this syntax is not fully defined, the mentioned algorithms cannot be used for automated comparison of models.

For this reason we decided to introduce a metametamodel which makes a clear separation between the <<instance-of>> relation between metametamodel and metamodels and the <<instance-of>> relation between metamodels and models. In fact, instead of the <<instance-of>> relation between metamodels and models a <<conforms-to>> relation is used to relate metamodels and models, and an <<instance-of>> relation in our approach relates our metametamodel and models. The interpretation of the <<conforms-to>> relation is as follows: A model conforms to a metamodel if each model element is related to only one metamodel element, and if relations between model elements reflect the relations between metamodel elements. In this way, the concrete syntax of models becomes clear, and it is possible to reason about matching model elements at a sufficient level of detail and formality.

2.2 New modeling framework

In this section we will first describe the overall architecture of our modeling framework. In particular, we will describe the new metametamodel which is the core of the framework. Then, we will describe the instantiation relation between the new metametamodel and metamodels, and we will describe an example metamodel. Next, we will describe the instantiation relation between the new metametamodel and models, and we will describe an example model. Finally we will discuss the differences and similarities between the developed framework and Ecore.

Note that, because the new modeling framework serve as the basis for describing all our other results, and all but the last chapter of this dissertation are slightly adapted versions of our published papers, the description of the new framework given in this chapter is replicated in more or less detail in all but the last chapter. However, the description of the framework given in this chapter is the most complete and encompasses all the other descriptions.

2.2.1 New metametamodel

The new, enhanced, metametamodel (EMMM) is depicted in Figure 2.2.

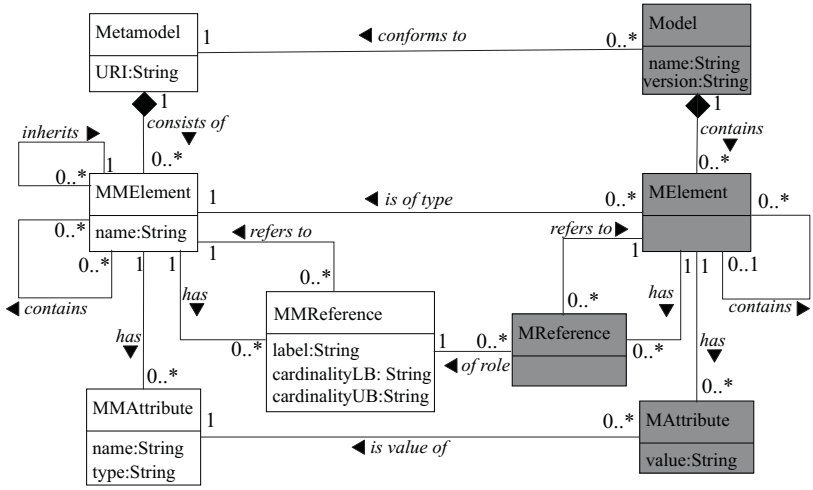

Figure 2.2: Enhanced metametamodel

Like MOF and Ecore, our enhanced metametamodel is depicted by using the object-oriented notions adopted from UML (classes and relationships). In particular, EMMM entities are represented by using the class notation. Thus, each entity has a name and has zero or more attributes. EMMM entities are related to each other by using the association notation (i.e. by connecting entities with lines). In EMMM two types of relations are used. The first relation type is the containment relation, which is represented by a line having a diamond at one end. The meaning of the containment relation is the following: the instances of the entity which is connected to the diamond (on the line representing the relation), *contain* the instances of the entity which is on the other end of the line. The term *contains* denotes that the contained elements must exist within one and only one container element, and if a container is deleted, the contained elements are also deleted. The minimum and maximum number of contained elements is specified by a cardinality attached to the end of the line opposed to the end of the line with the diamond. The second relation type is the common association relation. This relation denotes that zero or more instances of one entity are related to zero or more instances of an other connected entity. The cardinality attached to the end of the line connected to the entity denotes the minimal and maximal

number of instances of that entity that can be connected to an instance of the other connected entity.

However, unlike MOF and Ecore, where the meaning of the used notations was adopted from UML, the meaning of the concepts in EMMM is different than in UML. In particular, the entities labeled *Metamodel*, *MMElement*, *MMAttribute*, *MMReference*, and the relations between them, are used to define metamodels. Similarly, the entities labeled *Model*, *MElement*, *MAttribute*, *MReference*, and the relations between them, are used to define models. The relations starting from entities used to define models, and ending at entities used to define metamodels, are used to model the *conforms-to* relation between metamodels and models. In the next two sections, we will explain in detail the meaning of all EMMM elements.

2.2.2 Specifying Metamodels

A metamodel, in our framework, is an instance of an EMMM element named *Metamodel*. Each metamodel must have a unique identifier (i.e. a unique name), which is represented by an *Metamodel* attribute URI. At this moment, there are no mechanisms in place for enforcing the global uniqueness of the URI, but it is locally enforced in tools. Each metamodel contains zero or more instances of the metametamodel element named *MMElement*. Instances of *MMElement* represent domain concepts, and correspond to *classes* in traditional metamodels. Each instance of the *MMElement* can contain zero or more instances of the same element, thus opening the possibility for hierarchical metamodels. Each instance of the *MMElement* has a name, which must be unique in the set of all instances of the *MMElement* in one metamodel. Each instance of the *MMElement* contains zero or more instances of the *MMAttribute* element and zero or more instances of the *MMReference* element. The instances of the *MMAttribute* represent attributes of a domain concept represented by their containing *MMElement* instance. Each instance of the *MMAttribute* has a name (that must be unique in the set of all instances of the *MMAttribute* in a containing *MMElement* instance), and a type. The instances of the *MMReference* represent relations of a domain concept represented by their containing *MMElement* instance, to other entities. Each instance of the *MMReference* has a label (which must be unique in the set

of all instances of the *MMReference* of a containing *MMElement* instance), and a lower bound cardinality (cardinalityLB) and an upper bound cardinality (cardinalityUB) (such that $cardinalityLB \leq cardinalityUB$). The function of lower and upper bound cardinalities will be discussed in Section 2.2.3, since these cardinalities are related to models. Each instance of the *MMReference* must be related to one instance of the *MMElement*. This denotes the possibility of a relation between the model element that conforms to the instance of the *MMElement*, that the instance of the *MMReference* is contained in, and the model element that conforms to the instance of the *MMElement* that is referenced by the instance of the *MMReference*.

Example metamodel

An example metamodel is depicted in Figure 2.3.

The example metamodel allows for the definition of simple state-machines consisting of states and transitions. The metamodel contains an element which represent states (an instance of the *MMElement* named *State*), and an element which represents a transition (an instance of the *MMElement* named *Transition*). Each state has an attribute named *Label*, which serves as the label of the state in models. Transitions are not labeled, but each transition contains two instances of *MMReference*. Both of these instances are related to the instance of the *MMElement* representing a *State*. One of these instances is labeled *From* and represents the connection to the source (from) state of the transition. Another of these instances is labeled *To* and represents the connection to the target (to) state of the transition.

2.2.3 Specifying Models

In our framework, models are instances of an EMMM element named *Model*. Each model must be related to one metamodel by an instance of the relation labeled *conforms to* (i.e. a model conforms to a metamodel). Each model can contain zero or more instances of *MElement*. Each instance of an *MElement* contained in a model must be related to an instance of an *MMElement* in a metamodel that the model conforms to. Each instance of an *MElement* can contain

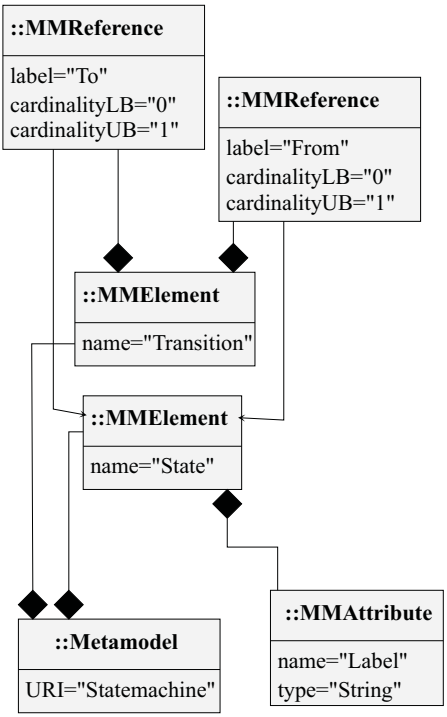

Figure 2.3: Example metamodel

other instances of an *MElement*. Each instance of an *MElement* contained in another instance of an *MElement*, must be related to an instance of an *MMElement* contained in an instance of an *MMElement* that the containing instance of the *MElement* is related to. Each instance of an *MElement* can contain zero or more instances of an *MAttribute* element, and zero or more instances of an *MReference* element. Each instance of an *MAttribute* contained in an instance of an *MElement* must be related to an instance of an *MMAttribute* that is contained in an instance of the *MMElement* that is related to an instance of the *MElement* that contains that instance of the *MAttribute*. Each instance of an *MAttribute* contains a value of that attribute. The name and type of the attribute can be inferred from the *MMAttribute* that the *MAttribute* is related to. Each instance of an *MReference* contained in an instance of an *MElement* must be related to an instance of an

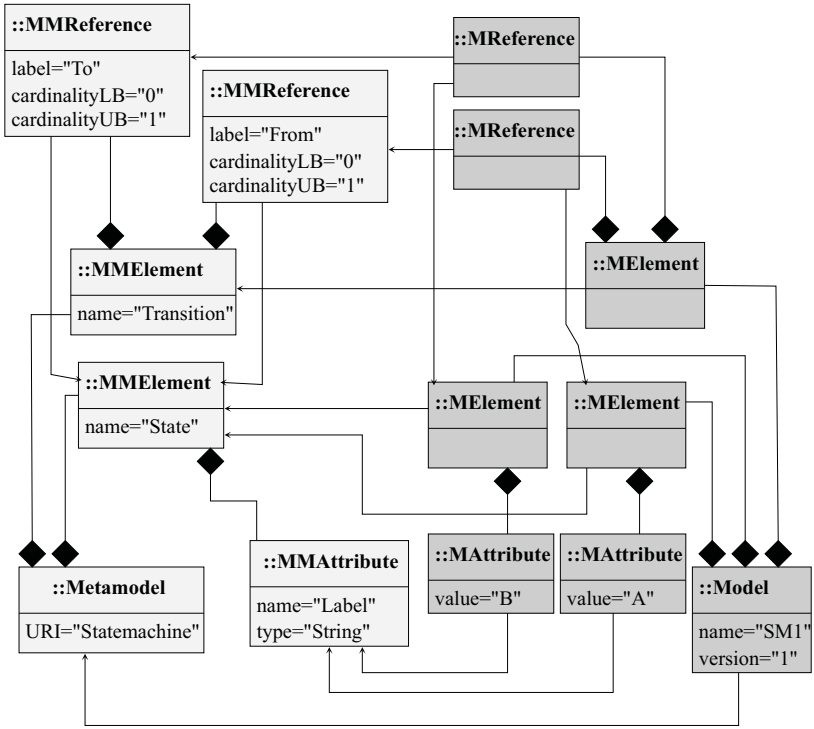

Figure 2.4: Example model

MMReference that is contained in an instance of the *MMElement* that is related to an instance of an *MElement* that contains that instance of an *MReference*.

An example model is depicted in Figure 2.4. Notice that the example model is represented together with the example metamodel, in order to emphasize their relation (model elements are colored dark-grey). The example model is a model of a state-machine containing two states and a transition. The first state is labeled *A*, and the second state is labeled *B*. The *From* reference of the transition refers to the state labeled *A*, and the *To* reference of the transition refers to the state labeled *B*. Thus, this transition has the state *A* as the source state. and has the state *B* as the target state.

2.2.4 Differences and similarities between EMMM and Ecore

Both EMMM (see Figure 2.2) and Ecore (see Figure 1.4) use similar concepts. For example, EClass in Ecore is analogous to MMElement in EMMM. The main difference between EClass and MMElement is that EClass instance can be abstract or interface, while this is not possible for MMElement instances. The reason for not including these concepts in MMElement is that abstract classes, or interfaces are important for defining metamodels, but are not explicitly included in models. Since we were interested predominantly in difference between models, we decided to exclude those concepts from metamodels that are not directly related to model differences. Thus, concepts like packages (represented by an EPackage class), factories (represented by an EFactory class), annotations, operations, and operation parameters, are also not included in EMMM.

EClass instances contain structural features, which are either EReference instances or EAttribute instances. EReference is analogous to MMReference, and EAttribute is analogous to MMAttribute. Both EReference and EAttribute instances have several attributes (attributes of EStructuralFeature and its ancestor interfaces) that are included, and have several attributes that are not included in MMReference or MMAttribute instances. The included attributes are MMReference label (corresponding to the name of an EReference), lower and higher cardinality bounds (corresponding to lowerBound and higherBound of an EReference), the name of the MMAttribute (corresponding to the name of an EAttribute), and the type of the MMAttribute (corresponding to the name of the type of the EAttribute).

The next difference is that EReference instances are used to model both association and containment relations in Ecore. However, in EMMM, containment relations are modeled by a special relation labeled *contains*. Because of this, all containment relations in EMMM are unordered and have $0..*$ cardinality. Again, since we were not interested in a "full-fledged" modeling framework, this constraint does not have much influence on models.

Furthermore, in Ecore there are pre-defined data types (taken from Java), while in EMMM there are no pre-defined data types, and the value of each attribute is

a string. Thus, the semantics of data types must be defined by the developers of the metamodels (or, similarly to Ecore, those developers can use Java data types, and inherit their semantics from Java).

2.3 A Metamodel for the definition of differences between models

In this section we will explain the details of the metamodel, that we developed, for the representation of difference between two models. This explanation should serve as a proof that the difference models, obtained by using our approach, fulfill all the requirements that the model differences should satisfy, as defined in Section 1.5.1.

Before going into the details of the differences metamodel, we will briefly explain the concept of a model difference. This discussion will be in context of the *state-based* approach to model differences. In this context, the difference between two models (e.g. the *original* and the *target*) is represented by another model, which is called *the difference model*. The difference model is calculated by a calculation algorithm that takes the *original* and the *target* model as arguments, and returns the differences between them: $differenceModel = calculate(original, target)$.

We consider the models as hierarchical acyclic graphs (i.e. trees). This is achieved by interpreting the model elements as nodes of the tree, and by interpreting containment relations between model elements as the edges between nodes of the tree. In order to calculate the difference between two trees, it is required to *match* the nodes in both trees that *model identical entities*. The matching process is part of the calculation algorithm and will not be discussed here, but its description can be found in Section 3.3.3.

After the nodes have been matched, the calculation of a difference is done by a process that traverses the tree representing the *original* model. The difference consists of a set of atomic differences, which are instances of elements in the differences metamodel. In order to define the types of all the possible atomic differences, we will describe the process of the difference calculation.

In the first step of the process, if the top nodes are not matched, then the difference is represented by a combination of both complete *original* and *target* models. Otherwise, the following recursive procedure is applied starting with the top nodes: For all matching nodes MN_1 in the original model and MN_2 in the target model, the difference between their attributes and references should be calculated first. Then, the difference between their children is calculated. All the children of the node MN_1 that are not matched to children of the node MN_2, are considered as *deleted* in the *target* model. All the children of the node MN_2 that are not matched to children of the node MN_1, are considered as *added* to the *original* model. All the children of the node MN_1 that are matched to children of the node MN_2, but that are not identical to their matches, are considered as *changed* in the *target* model. All the children of the node MN_1 that are matched to children of the node MN_2, and are identical to their matching elements, are considered as *unchanged* in the *target* model.

Based on the specified difference calculation process, it can be concluded that there are three main types of atomic differences between two models: *added model elements*, *deleted model elements*, and *changed model elements*. However, the described calculation process does not take into account the possibility of *moved* model elements. A *moved* model element is a model element in the original model that models the same entity as a model element in the target model, but the parents of those model elements do not match. Since the moving of model elements is a real possibility, especially in manually designed models, the fourth type of possible atomic differences are *moved model elements*.

The model differences metamodel is depicted in Figure 2.5.

Differences models are instances of the *Differences Model* element. All differences models reference two models, denoted by the references labeled *original* and *target* connecting the *Differences Model* element and the *Model* element. The models referenced by the *original* and the *target* reference are the models that have been used in the calculation of the differences represented by the differences model. Each differences model contains zero or more instances of the *ElementDifference* element. Instances of the *ElementDifference* element represent added, deleted, changed, or moved model elements. In particular, the element *AddedElement* represents added elements, the element *DeletedElement*

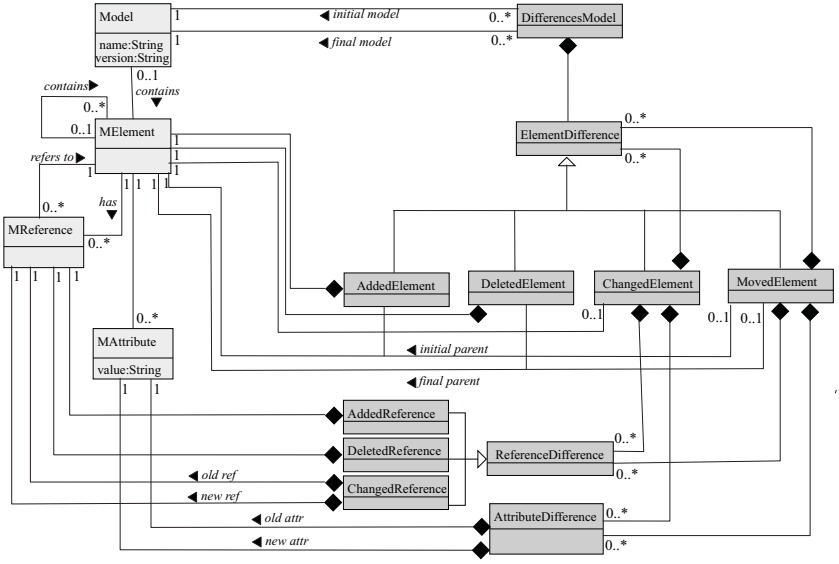

Figure 2.5: Model differences metamodel

represents deleted elements, the element *ChangedElement* represents changed elements, and the element *MovedElement* represents moved elements.

Each instance of the *AddedElement* contains one instance of the added *MElement*. This contained instance of the *MElement* is the complete added element. The containment relation is used instead of the association relation to satisfy the *self-contained* requirement (if the association relation would have been used instead, the differences model could not have been used without the model that it would have been associated with). Each instance of the *DeletedElement* contains one instance of the deleted *MElement*.

Each instance of the *ChangedElement* contains zero or more instances of the *ElementDifference*. In this way the hierarchy is implemented in the differences models. Furthermore, each instance of the *ChangedElement* references a matched *original* model element and a matching *target* model element. This is required in order to fulfill the *Invertible* requirement, because in this way it is possible to revert back to the *original* model by using the *target* model and the differ-

ences model. Each changed element can also contain zero or more instances of the *AttributeDifference* element and zero more instances of the *ReferenceDifference* element. An instance of the *AttributeDifference* element represents changes to one of the attributes of the model element. For each changed attribute there can be only one instance of the *AttributeDifference* element. For reasons of reversibility, each *AttributeDifference* instance contains the values of the attribute in both the *original* and the *target* model. An instance of the *ReferenceDifference* element represents changes to references of the model element. Since the number of references of a model element is not pre-defined, it is possible to add references, delete references, or change references. This is represented by *AddedReference*, *DeletedReference*, and *ChangedReference* elements.

2.3.1 Model Differences Example

In this subsection we discuss an example differences model. The example differences model is obtained by a calculation algorithm, specified in Section 3.3.3, that considers the model depicted in Figure 2.4 as the *original* model, and considers the model depicted in Figure 2.6 as the *target* model. The *original* model is a state machine containing two states labeled *A* and *B*, and a transition from state *A* to state *B*. The *target* model has one more state, labeled *C*, and this state is connected to the state labeled *A*. Furthermore, in the *target* model the state labeled *B* has a different label - *BA*.

The resulting differences model is depicted in Figure 2.7. Note that the contents of the differences model depend on the calculation algorithm, and the depicted differences model is just one of the many possibilities.

The light-gray elements in the example differences model represent elements that are either an instance of the *Differences Model* element, or are contained, directly or indirectly in those elements. The dark-gray elements in the example difference model represent elements that are parts of the original or the target model, and as such must conform to a metamodel that those two models conform to. This is the case even if the dark-grayed element is contained in the light-grayed element, as is the case with the instance of the *AttributeDifference*.

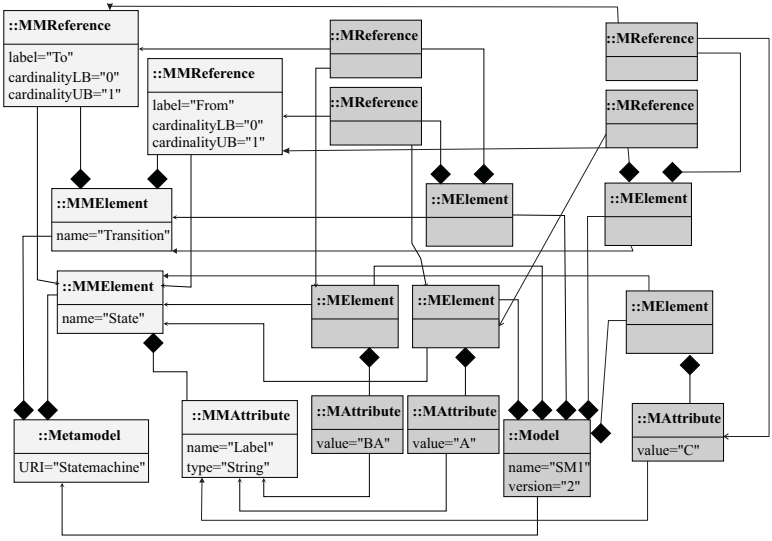

Figure 2.6: A new version of the example model depicted in Figure 2.4

2.4 Conclusions and Future work

In this chapter we discussed an instantiation problem of existing layered approaches to setting up a modeling infrastructure, i.e. a metametamodel. The cause of this problem is the fact that the instantiation relation cannot be used to connect more than two modeling layers, unless a transformation is performed on the results of the instantiation to make them again instantiable. The consequence of this problem is that the model differences cannot be represented in the required level of details. To alleviate this problem, we have defined our own metametamodel. By using our metametamodel we have defined a metamodel for the representation of model differences that shows all the details of model differences. The comparison of our approach and the most popular approach to represent differences between MOF-based models is given in more detail in section 3.2.

However, our metametamodel was designed to be minimal and elegant, and it was not designed to be a full-fledged modeling framework. Thus, it would be interesting, as future work, to expand our metametamodel to include modeling

concepts (such as packages, operations, diagram types, etc.) used in more popular formalisms such as MOF or Ecore.

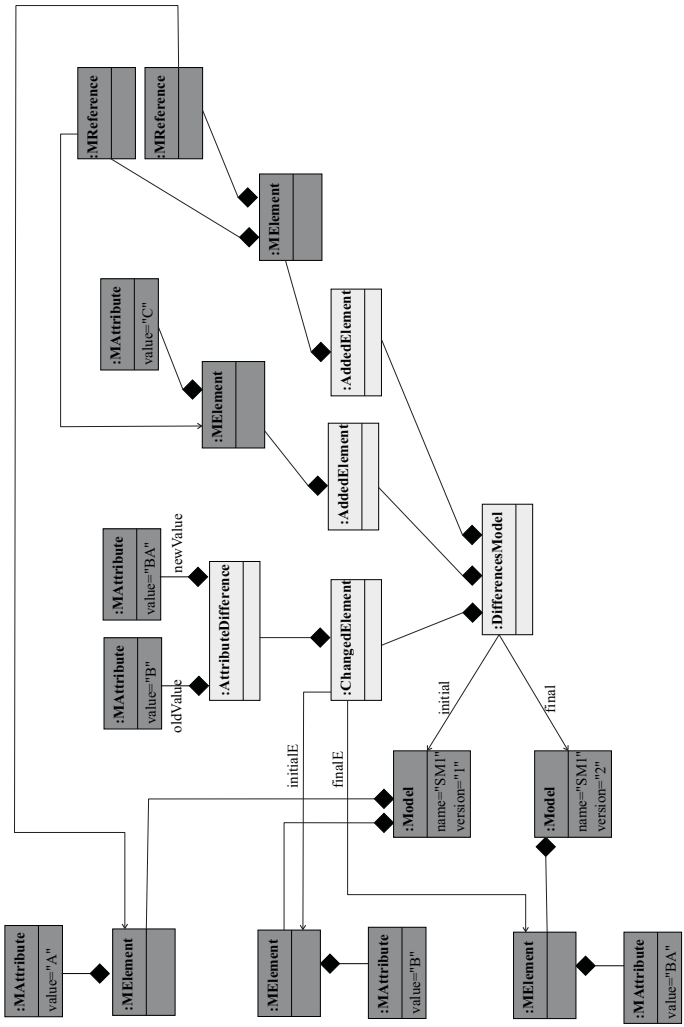

Figure 2.7: Example differences model

Chapter 3

Model Differences Representation and Calculation

This chapter describes our research in representation and calculation of state-based model differences. The main goals of this research were to improve the quality of existing methods for the representation of model differences, and to improve the quality of existing approaches for the calculation of model differences. In order to achieve the highest clarity in representing model differences, we introduced a new metametamodel. By using this metametamodel we created a metamodel for model differences that discloses important facts of model differences, that were not discussed in any of the existing approaches. Our algorithm for the calculation of model differences is based on existing state-of-the-art differences calculation algorithms. However, while all other algorithms leverage only one type of model matching, in our algorithm all recognized types of model matching can be used. This was achieved by designing our calculation algorithm in

parallel with the special configuration metamodel, that allows the users of the algorithm to configure it to use the type of model matching that gives the best results for a particular metamodel.

3.1 Introduction

Model Driven Software Engineering (MDSE) is a field of Software Engineering that focuses on models as main design artifacts, and imposes model transformations as means of relating models. Consequently, mature model configuration management systems are required to manage the complexity of modeled systems in MDSE environments. One of the major functions of model configuration management systems is model comparison. Model comparison (or differencing) is a complex process, which involves at least three concerns: representation, calculation, and processing of differences [84]. The rationale behind this separation of concerns is that usually it is not only required to calculate differences, but it is required to store, process, and visualize them in the context of a model configuration management system. However, most of the existing approaches to model comparison deal with only one aspect of model comparison, and thus make it hard to integrate all aspects into one approach. Moreover, there are only a few tools that can be used as a part of a model configuration management system to compare models. The reason for this is that these tools are either metamodel-specific, meaning that they can be used to compare only models conforming to one metamodel (e.g., UMLDiff [123]), or that these tools are framework specific (e.g., EMF Compare [9]).

The goal of the research whose results are presented in this and in chapter 5, was to specify a methodology that integrates all three aspects of model comparison, and to build a tool that is framework-independent, that allows metamodel-independent model comparison, and that can be used stand-alone or in a context of a configuration management system. In order to fulfill this goal, we have first explored existing approaches to representation, calculation and visualization of model differences. Next, we have selected state-of-the-art approaches in all of the explored aspects, and we have extended those approaches such that our goal would be fulfilled.

In order to use the model differences in the context of MDSE the representation of differences should satisfy the following set of requirements, initially defined in [49]:

- Model based: The differences should be represented by a formal difference model.
- Minimalistic: The differences should contain a minimal number of objects.
- Self-contained: The differences model must contain all the information autonomously without relying on data contained in the compared models.
- Transformative: It should be possible to transform one model into another model using their difference model.
- Invertible: It should be possible to revert back to the initial model using the transformed model and their difference model.
- Compositional: The result of subsequent or parallel modifications is a difference model whose definition depends only on difference models being composed and is compatible with the induced transformations.
- Metamodel independent: The difference metamodel should be independent of a particular metamodel (e.g. UML).
- Layout independent: The difference metamodel must be agnostic of the presentation issues.

The complete rationale behind these requirements can be found in [49]; we will just give a short explanation of why these requirements are needed. The differences should be model based and transformative in order to be used in MDSE environments. The self-contained and minimalistic requirements are related, in a sense that they capture the idea that the differences should contain all the changes and only the changes. The differences should be invertible in order to allow the revert operation, which is commonly used in configuration management systems. They should be metamodel and layout independent because they should be usable in generic environments to allow the building of domain specific comparison frameworks.

Traditional approaches to the representation of model differences, by means of difference models [50, 85, 89], do not completely fulfill all the specified re-

quirements. For example, in the approach presented in [50], for each (MOF-based [20]) metamodel a new difference metamodel is generated. However, the generated difference metamodels are too coarse, in the sense that the elements of the difference metamodel are related only to complete metamodel elements, and not to their parts. For example, changing one attribute of an object requires the inclusion of the complete object in the difference model. Thus, the difference models can contain unnecessary information, because usually changes do not affect the complete model element but only parts of it. Also, changes to instances of relations between metamodel elements are not considered in the difference metamodel. In the approach of [89], which is based on GME [17], models are considered as graphs, and the difference metamodel is also graph based. However, their difference metamodel is designed such that the information related to the change of edges is not contained in difference models (e.g. only added or deleted edges are considered), and it lacks in handling of ambiguous models (models in which an element can contain two or more identical sub-elements). In the approach of [85], which is based on Ecore [11], a model-independent difference metamodel is specified. However, the specified difference metamodel is not invertible because it does not consider added or deleted model elements.

A solution to the aforementioned shortcomings of the mentioned approaches is to define only one difference metamodel that is able to capture all the differences at appropriate granularity and without containing unnecessary information. In order to do this we deviate from the usual *four-layer* metamodeling architecture. We introduce an enhanced metametamodeling architecture that offers a different organization of layers and relations between them. The introduced architecture can be considered a *domain specific metametamodeling architecture* [125] which is specialized in describing model differences, unlike traditional metametamodeling architectures, like MOF and Ecore, which are specialized for modeling. However, the results we obtained by using this architecture are applicable to other architectures as well. By using this enhanced metametamodeling architecture, we define a difference metamodel that satisfies all the specified requirements. Section 3.2 gives a more detailed description of our approach for representing differences.

After the representation format has been set, the calculation process can be specified. The calculation process takes two models as arguments and returns their

difference model conforming to the specified difference metamodel. Traditional approaches to difference calculation consider models as trees, and use the structure provided by these trees to guide the calculation process [39, 79, 83, 89]. The calculation algorithms used for calculating differences are based on *matching* elements in one model to elements in another model by using one of the four recognized types of matchings: static-identity, signature-based, similarity-based and language-specific [84]. There are two traditional approaches to specifying the calculation algorithm—imperative and declarative. An example of an imperative specification is work presented in [83], where a special language is used to define the matching criteria for two models of the same metamodel. However, imperative approaches, although more powerful, since the user can define all the details of the matching, are less generic, since for each new metamodel the comparison algorithm needs to be rewritten. Thus, we focus on declarative approaches which are more generic. However, most traditional declarative approaches have a fixed calculation algorithm and use a specific type of matching. For example, in [39] a four-step top-down algorithm for calculating model differences based on static-identities of model elements is presented. In a differences calculation approach by [89] a two-step top-down algorithm is presented. In the first step a signature-based matching is used to map elements of one model to elements of the other model. In the second step the similarity-based matching is used to improve the results of the first step by matching elements, not matched in the first step, based on the similarity of their relations. In the approach proposed by [79] an algorithm for calculating model differences using similarity based matching is presented. This is a bottom-up algorithm, which has two steps, and is based on a tree-comparison algorithm of Chawathe et al. [48]. In the first step, model elements are matched based on their similarities, and in the second step the differences are calculated by taking into consideration matched elements. Unlike the other two mentioned algorithms, the calculation process of this algorithm can be influenced by setting the threshold values for a similarity function.

Our approach to calculating differences extends the approach of Keller et al. [79]. We chose to extend this approach because it uses similarity-based matching, that is the most generic metamodel-independent approach to model matching, and because it was presented in the context of MDSE, and thus was easy to extend and adopt to our purposes. The core of our approach is a comparison

metamodel that all the models being compared must conform to. This metamodel allows us to represent models as trees, thus enabling us to use the ideas of tree-comparison [48] in our algorithm. Although our algorithm is based on similarity-based matching, the comparison metamodel, the similarity function, and the comparison algorithm are designed in such a way that it is possible to use all four types of matching. For example it is possible to define external functions which can be used to match similar elements of a specific type, thus enabling the signature-based matching. Also, it is possible to declare a set of attributes as a carrier of the unique identifier of model elements, thus enabling the static-identity matching and the language-specific matching. Section 3.3 gives a more detailed description of our approach to calculating differences.

Based on the presented approaches we have created a prototype tool for representation, calculation and visualization of differences which can be found online in [30].

3.2 Representation of Model Differences

As already mentioned, traditional four-layer metamodeling approaches are limited in describing differences metamodels which fully satisfy the model difference requirements introduced in Section 3.1. In order to illustrate this, we provide a description of the approach of Cicchetti et al. [50], which is a state-of-the-art approach in the representation of differences between MOF-based models. The schematic representation of their approach is depicted in Figure 3.1. In their approach, for each MOF-based *Metamodel* a specific *Differences Metamodel* is generated. Particulary, as depicted in the top right part of Figure 3.1, for each Metamodel element, three elements of the Differences Metamodel are created (Added, Deleted, and Changed element). For each selected Metamodel element the instances of the created elements represent added, deleted or changed instances of the selected Metamodel element.

Next, we will consider the UML differences metamodel (see Figure 3.2) which is specified in [50]. Based on this metamodel, the following two claims can be made: The first claim is that the *ChangedOperation* element of the differences metamodel specializes and references an *Operation* element of the UML meta-

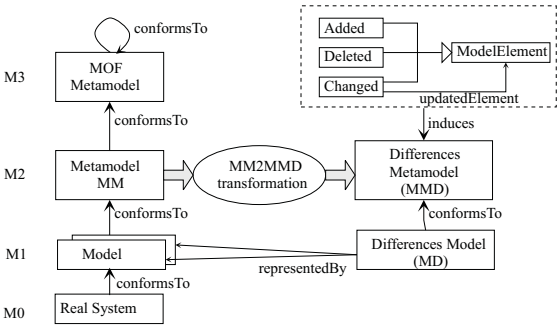

Figure 3.1: Schematic of an approach to obtain differences metamodel from a metamodel presented in [50]

model. As a consequence, if an instance of an *Operation* element has changed, the difference model will contain both the instance of the *Operation* element and the instance of the *ChangedOperation* element. Thus, even if the change has happened to only one attribute of the instance of an *Operation* element, the complete instance of the *Operation* element must be included in the differences model. Furthermore, in models with deeper hierarchies, a change to an attribute of an instance of a metamodel element deep in the hierarchy induces the inclusion of all the ancestors of this element instance in the differences model. The second claim is that the relations between metamodel elements are not transformed. As a consequence, the instances of these relations are not directly included in the difference models. For example, if an instance of a *Class* element changes its parent, it is not specified in the differences metamodel how this change is treated. Based on these claims, it is clear that the difference metamodels obtained by using the approach of [50] do not fully satisfy all of the requirements specified in the introduction, in particular minimality and self-containment.

The cause of this shortcoming is the fact that traditional metametamodels (MOF in this case), in order to be generic, equate *is-instance-of* and *conforms-to* relations between the metametamodel, metamodels and models. Thus, models conform to, and are considered the instances of metamodels. Consequently, metamodels are at the appropriate ”level” for defining difference metamodels. However, since the metamodels are instances of metametamodels, in order to

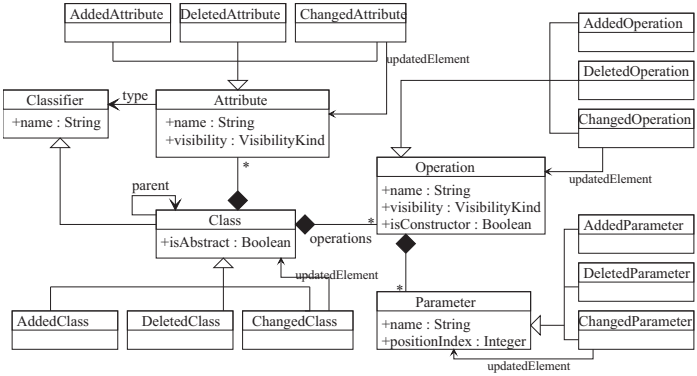

Figure 3.2: UML differences metamodel as defined in [50]

use metamodels to instantiate models they need to be interpreted. This interpretation abstracts from the metametamodel details that are important for defining differences at a granularity required to satisfy the minimality requirement. For example, it is not possible to relate elements of the differences metamodel to parts of the elements present in the metamodel (parts such as attributes and relations), but the elements of the differences metamodel are related to complete metamodel elements. Also, the important information on the relations between metamodel elements is hidden behind their interpreted forms, which are edges connecting metamodel elements.

In order to specify a differences metamodel which fully supports the requirements defined in Section 3.1, and in order to shed more light on the meaning of model differences, we introduce a new metametamodel architecture.

3.2.1 Enhanced Metametamodel used to describe fine-grained differences metamodels - EMMM

The enhanced metametamodel is simple enough to allow an effective representation of model differences, and it is powerful enough to describe all graph-based models. This metametamodel also induces a new organization of the standard four-layer architecture. The new architecture is depicted in Figure 3.3, and the

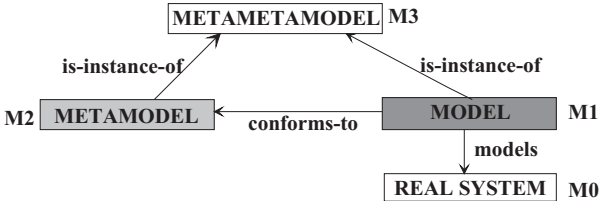

Figure 3.3: New organization of the layered architecture of metamodels and models

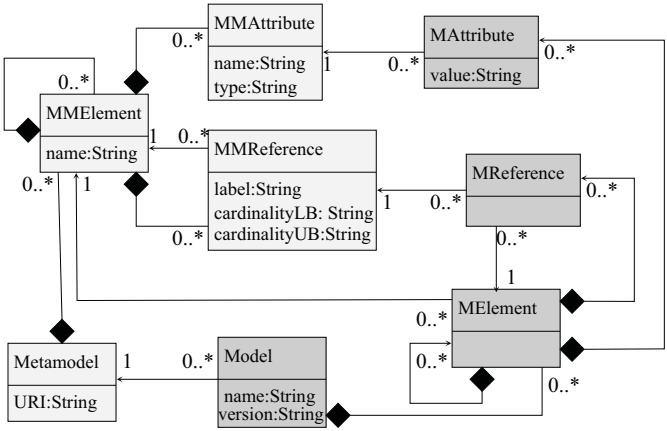

Figure 3.4: Enhanced metametamodel - EMMM

enhanced metametamodel (EMMM) is depicted in Figure 3.4. A more extensive discussion on the reasons for introducing the enhanced metametamodel, and on the syntactic and semantic details of the enhanced metametamodel is given in Chapter 2.

In the traditional *four-layer* architecture, a model is obtained by first instantiating the metametamodel into a metamodel and then instantiating the metamodel into a model. Thus, it uses two instantiations in sequence. These instantiations are of a different nature, since they start from different kinds of models and result in different kinds of models. This is comparable to a *two-level grammar*. In the new architecture, both the metamodel and the model are obtained by instantiation from the metametamodel. Here, there is only one notion of instantiation, which

is applied 'in parallel'. This is comparable to a single programming-language grammar that defines both type constructions (cf. metamodels) and typed expressions (cf. models).

Our enhanced metametamodel explicitly captures the relation between metamodels and models on one level. The interpretation of the enhanced metametamodel is as follows: The metametamodel describes both the metamodels and the conforming models. Metametamodel elements that are related to metamodels are colored light-grey, and those elements that are related to models are colored dark-grey. Metamodels are instances of the *Metamodel* element, and models are instances of the *Model* element. Metamodels contain instances of the *MMElement*. Models contain instances of the *MElement*. Each *MElement* instance is related to one *MMElement* instance (similarly for *MReference* and *MAttribute*). Instances of *MElement* correspond to *Objects* and instances of *MMElement* correspond to *Classes* in the standard object-oriented paradigm.

An example metamodel and a model conforming to that metamodel are depicted in Figure 3.5 (*A*::*B* expresses that *A* is an instance of metametamodel element *B*). The example metamodel allows the definition of State Machines. The elements constituting the metamodel are colored light-gray. The metamodel contains two instances of the *MMElement*. One of them, named *State*, represents states and the other one, named *Transition*, represents transitions. The *MMElement* instance representing states contains an instance of an *MMAttribute* named *label*. The *MMElement* instance representing transitions contains two instances of *MMReference*, which both reference an *MMElement* instance representing states. One of these *MMReference* instances denotes the end of the transition connected to the source (*From*) state, and the other *MMReference* instance denotes the end of the transition connected to the target (*To*) state.

The example model is named *SM1*, and its elements are colored dark-gray. *SM1* contains three instances of the *MElement*. Of those three instances, two are states and one is a transition (since two of them reference *MMElement* instance named *State* and one of them references *MMElement* instance named *Transition*). One of the states contains an *MAttribute* instance having the value *A*, and the other state contains an *MAttribute* instance having the value *B*. This means that the labels of those states are *A* and *B* respectively. The transition contains two in-

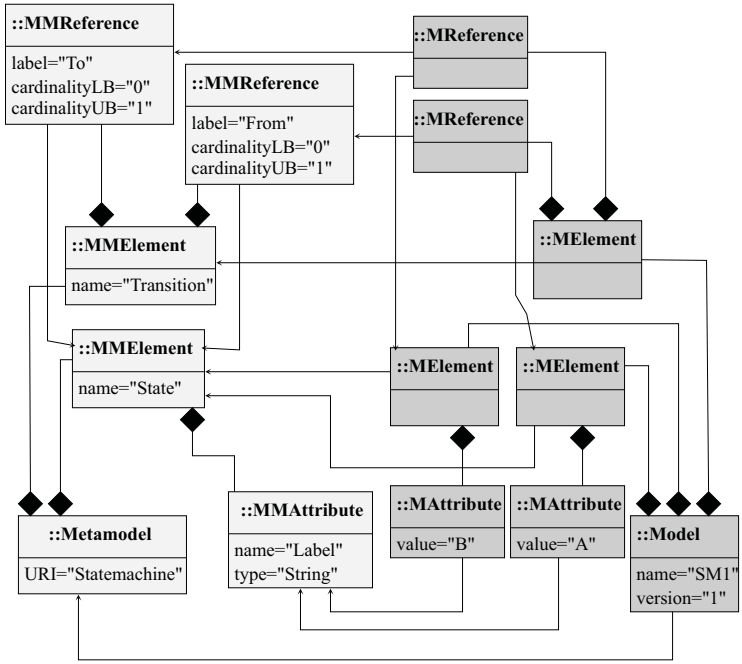

Figure 3.5: Example metamodel and model

stances of the *MReference*. One of those instances is associated to *MMReference* instance labeled *From* and it references a state labeled *A*. Thus, state *A* is the *From* state. The other instance of the *MReference* is associated to *MMReference* instance labeled *To* and it references state labeled *B*. Thus, state *B* is the *To* state.

Our enhanced metametamodel (and its associated architecture) makes the relation between models and metamodels more explicit than the traditional *four-layer* architectures. This also allows for a fine-tuned representation of differences metamodel, which we describe in the next section.

3.2.2 Differences metamodel

To describe our differences metamodel, we will first define the structure and meaning of differences between two models. We will assume that there are two models A and B being compared, that both of these models conform to the same metamodel, and that both are instances of the EMMM. Then, model A and model B each consist of a set of *MElement* instances (we will use 'object' as a synonym for '*MElement* instance' in this section). Within the set of objects of model B there could exist objects which represent the same entities as objects in model A. We will say that those objects *match* objects in model A. The mechanism to check if two objects from two different models *match* will be discussed in Section 3.3.

Since one of the requirements for model differences is that they are transformative, we will consider the differences between the models A and B as a minimal set of objects such that it is possible to use that set of objects to transform model A into model B by model transformations. Since models A and B consist of sets of objects, we consider the following set operations as a model transformation that transforms model A into model B', equal to model B:

- Delete all objects in the set of objects of model A that are not matched by any object in the set of objects of model B, obtaining the model B'.
- Add all objects to the model B' that are in the set of objects of model B that are not matched by any object in the set of objects of model A.
- For each object O_A in model B' that has a matching object O_B in model B, transform the attributes, references and sub-objects of object O_A such that the resulting object is identical to object O_B. If two matching objects are already identical, do nothing.

The last operation is specified to satisfy the minimality requirement, because although only the first two operations can be used to transform one set of objects into another set of objects, the resulting difference models would contain unnecessary information.

Our differences metamodel is created as an extension of EMMM to support the specified requirements and set operations. The relation of the differences meta-

model and the new architecture is depicted in Figure 3.6, and the metamodel itself is depicted in Figure 3.7.

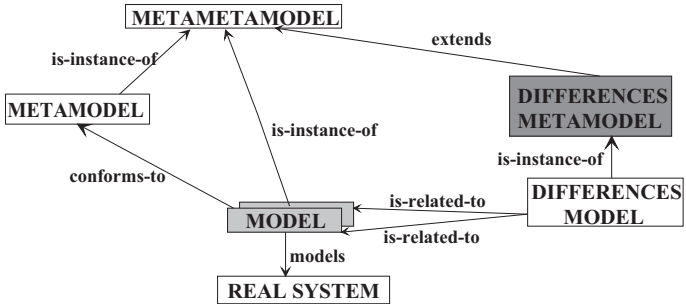

Figure 3.6: The position of the differences metamodel the new architecture

The interpretation of the differences metamodel is as follows: The difference models are instances of *DifferenceModel*. The difference models contain the instances of *ElementDifference*. The objects that should be deleted from the set of objects of model *A* are represented by an instance of a *DeletedElement*. The objects from model *B* that should be added to the resulting set are represented by an instance of an *AddedElement*. The *ChangedElement* contains the differences between attributes, references and sub-objects of objects in model *A* and matching objects in model *B*. The differences contained in a *ChangedElement* are sufficient to transform an object from model *A* to its matching object in model *B*. Notice that in order to fulfill the invertibility requirement, the relations between difference elements and model elements are containments, and not references.

Our representation gives a clearer picture of what differences really are, viz. sets of difference objects that are used in a model transformation to transform (the set of objects representing) one model into (the set of objects representing) another model. Our differences metamodel is truly metamodel independent, since the elements that represent differences are only related to metametamodel elements representing model elements. This representation is fine-grained and solves the shortcomings of the approach of [50], because the instances of our difference metamodel contain only the actual changes to (the parts of) objects of the model, and not complete objects. Thus, difference models contain complete objects only

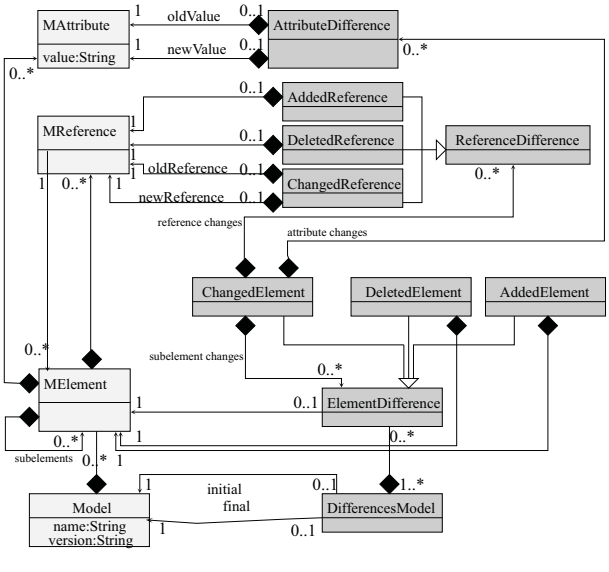

Figure 3.7: Differences metamodel

when those objects are added or deleted, and contain only the changed parts of changed objects. Also, this representation treats the relations between model elements in the same way it treats other parts of model elements.

3.3 Calculation of Differences

The process of calculating model differences usually consists of three parts: representing the two models being compared as trees (i.e. exposing the tree-like structure of models), *matching* the nodes in those trees, and calculating differences based on the matchings found [39, 79, 89]. Representing models as trees is possible because traditional metametamodels allow a hierarchical representation of models. The *matching* of the nodes in model trees has the goal of finding objects that represent the *same* entities in both trees. There are four general approaches to matching [84]. Static-identity based matching assumes the presence of universally unique identifiers (UUID) that are assigned to model elements

upon creation and that persist with the model elements. In this approach, model elements that have identical UUIDs are matched.

Signature-based matching assumes that for each model element, a uniquely identifying *signature* can be calculated based on features of the model element. The elements that have the same signature are matched.

Similarity-based matching requires a similarity function that calculates the similarities between two model elements. Only elements with similarity value greater than a certain threshold value are matched.

Language-specific matching assumes that the metamodel offers a way of uniquely identifying model elements. For example, in Ecore [11], some attributes of a model element can be marked as *keys*. Two instances of the model element in Ecore can be matched if they have identical values of the attributes marked as keys.

Our approach to calculating differences extends the approach of Kelter et al. [79], which utilizes similarity-based matching. In their approach, two models being compared are represented as trees. Their calculation algorithm has two steps. In the first step the model trees are traversed bottom-up, and *similar* model elements in both trees are *matched* by using a similarity function. In the second step the model trees are traversed top-down and based on the matchings discovered in the first step, the differences are calculated. However, the similarity function provided is not sufficiently generic. For example, the matching process cannot be influenced except by setting the threshold value of model element types. Also, certain structural properties of the models being compared, particularly type information, are not considered.

In our opinion, the similarity function, and with it the comparison process, should be more controllable. For example, there should be the possibility to define similarity sub-functions for attributes of model elements. In this way, a domain expert could tune the calculation process such that domain specific matching is obtained. To allow a more configurable calculation algorithm and to allow the inclusion of the additional structural properties in the calculation algorithm, we extended the metamodel specified by Kelter et al. [79]. The extension is such that our new metamodel allows not only similarity-based matching, but also other

types of matching. Since we use the enhanced metametamodel to represent the differences metamodel, our calculation metamodel is specified in conformance to the enhanced metametamodel, and is depicted in Figure 3.8. The calculation

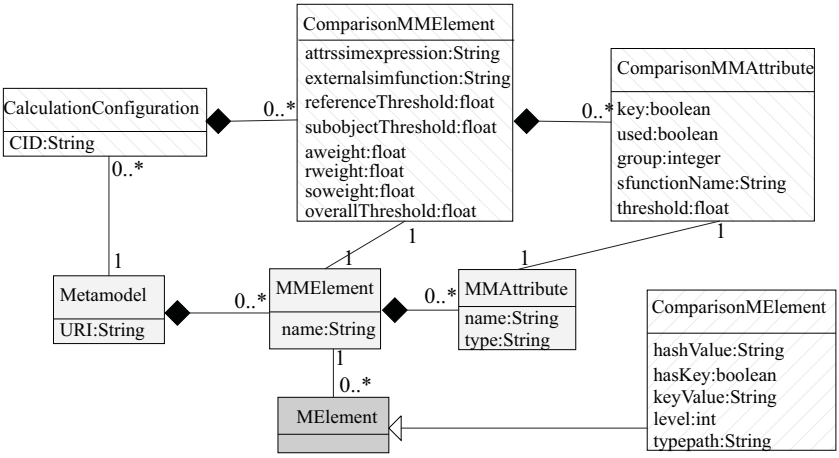

Figure 3.8: Calculation metamodel used in our approach to calculating differences

metamodel allows the representation of models as trees, and consists of two (logically) separated parts. The first part is represented by the *CalculationConfiguration*, *ComparisonMMElement* and *ComparisonMMAttribute* elements. This part is related to the metamodel, and can be considered as a metamodel-specific configuration of a difference calculation process. For each metamodel element a domain expert can relate an instance of a *ComparisonMMElement* (and within it instances of a *ComparisonMMAttribute*), and thus configure the comparison algorithm in order to ensure the best matching of model elements conforming to that specific metamodel element. Notice that although the model differences are metamodel independent, the calculation process is metamodel dependent because the comparison algorithm exploits the structure specified by metamodels.

The second part is represented by the *ComparisonMElement*, and is related to the models being compared. This part contains information such as the hash-value of a model element, or the existence of universally unique identifiers (called

Keys here) in certain model elements. The instance of *ComparisonMElement* is generated in a preprocessing step for each model element in each model being compared, and is used instead of a model element as a node in the model tree.

3.3.1 Preliminaries: Tree-comparison algorithms

In [124] an algorithm is given for calculating the best *edit distance* between a string and a tree (the edit distance is a minimum-cost mapping between two trees). The authors prove that the problem of calculating the edit distance between two unordered trees is in general NP-complete. The cause of this is the lack of structure of the trees being compared. However, if a specific structure is imposed upon the trees being compared, polynomial time algorithms exist as shown by Liu et al. [90] for unordered trees, or as shown by Chawathe et al. [48] for ordered trees (*LaDiff* algorithm).

The algorithm that we define, however, does not calculate the minimum edit distance as defined in [124]. Because of the extra structural properties imposed by the models we use in the calculation process, the minimum edit distance cannot be used in its original form. The main reason for this is that the additional structural properties allow only the deletion of a complete sub-tree, thus disallowing replacing a node with its sub-tree. Another reason for not calculating minimum edit distances is that the model differences should help in understanding the evolution of models. Minimum edit distances, in the mathematical sense, could impose difference models that are not understandable because the elements that do not represent the same entities in both models would be included in differences. Thus, in our approach we have two related sub-goals: finding minimal differences, such that they provide the most insight into the (possible) evolution of one model into another model.

3.3.2 Preliminaries: Assumptions and Definitions

We assume the existence of two models, A and B, having the same metamodel, and conforming to the EMMM. The requirement is to calculate the differences between A and B such that those differences conform to a differences metamodel

specified in Section 3.2.

Our algorithm for calculating differences consists of three steps. In the first step the similarities between objects in models A and B are calculated. In the second step, based on the similarities found, a matching of objects is calculated. In the last step the calculation of differences based on the calculated matchings is done.

Some approaches, for example [48] or [79], combine the first and the second step into one step where the similarities are calculated and similar objects are immediately matched. We kept these steps separate, because the structure of trees representing models and the characteristics of our similarity function then allow better matchings to be found.

There are four structural properties of the models being compared that are exploited in our calculation algorithm to improve its performance and precision. The first property is that the trees representing models (referred to as comparison trees in the remainder) will be composed only of instances of a *ComparisonMElement* element (thus, only objects will appear as nodes in comparison trees). The attributes and references of objects will not be considered as nodes in the comparison tree (though they will be used in the calculation). Since elements in the model are considered nodes in comparison trees, we use *node* as a synonym for *model element* from now on.

We will say that $t(x)$ is a type of a node x, $par(x)$ is a parent of x and $an(x)$ is an ancestor of x. The ancestor relation is the transitive closure of the parent relation. Relation $m(x,y)$ expresses that objects x and y match. Finally, $s(x,y)$ expresses that objects x and y are similar.

The second property involves the structure provided by the metamodel: if an object x in model A matches object y in model B ($m(x,y)$), then $t(x) = t(y)$. This property states that two objects can match only if their types coincide. The type of an object is the *MMElement* instance that that object is related to. This property is used in the first step of the algorithm to improve performance, by matching only objects of the same type.

The third property reflects our view on the relation between similar and matched objects, in case that moved objects are not allowed: if object x in model A

matches object y in model B, then $m(an(x), an(y))$, and $t(x) = t(y)$, and $s(x,y)$ holds. This property states that two objects can match if their ancestors match, if they are of the same type, and if they are similar. It is used in the third step of the algorithm to match similar elements.

The final property is that all objects in both models are identifiable. Thus, each object has a *locally* unique identifier, which is unique in the model that the object resides in. For example, this unique identifier can be the address of the memory location of the object.

A detailed description of the comparison algorithm can be found in the following Section.

3.3.3 Model Comparison Algorithm

This section provides further details of our comparison algorithm.

In order to calculate similarities between two objects, the meaning of the similarity between two objects needs to be defined first. We will say that two objects are similar if they can be considered the *same* entity in both models. The *same* in this context means that the designer of the second model considers the entity in the second model as an evolved version of an entity in the first model. Thus, the *same* entities do not have to be identical.

One way to define similarities is by using a similarity function. The similarity function defined in [79] takes two model elements (also referred to as objects), and returns a real value between 0 and 1. If the similarity value between two objects is above a certain threshold, then those objects are considered the same.

In [79], the similarity function is calculated between all model elements that are on the same level in the first and the second model. This approach to calculating similarity is fast enough for smaller models, but for larger models it poses a bottleneck. For larger models, which contain thousands of objects, with large number of attributes, this algorithm, although of polynomial complexity, requires an excessive amount of processing time. The approach presented in [108] alleviates the problem by using results from the field of multidimensional search. In their

approach, each attribute is considered as one dimension in a multidimensional space, and a set S_B is transformed into a structure which allows fast discovery of the most similar objects to the objects in this set. However, these results can only be applied in cases where all attributes are related, and when all attributes can be transformed into a metric space. In general this is not the case, thus we have decided to use a different format of the similarity function.

We define the similarity function as a function that returns a value between *0* and *1*, but for each metamodel element a threshold is set. In line with our definition of the *same* entities, if two model elements (objects further on) can be considered the *same*, then the similarity between them is larger than the defined threshold.

In order to give a formalized definition of our similarity function, we must consider parts of objects. As defined by the enhanced metametamodel, each object may contain attributes, references, and sub-objects (attributes are instances of the *MAttribute* element, references are instances of the *MReference* element, see Figure 3.4). We introduce separate similarity sub-functions for attributes, references, and sub-elements, and the results returned by those similarity functions are used to calculate the final similarity value between two objects. In particular, for each object that is able to match object X, three similarity values concerning object X are kept, one for attributes, one for references, and one for sub-objects. Separate similarity functions are introduced because attributes, references, and sub-objects are conceptually different, and thus it is safe to do separation of concerns at this level. Let $S_{attributes}(X)$ be a set of objects that are similar to object X considering attributes. Let $S_{references}(X)$ be a set of objects that are similar to object X considering references. Let $S_{subobjects}(X)$ be a set of objects that are similar to object X considering sub-objects. Then, for all distinct objects Y in sets $S_{attributes}(X)$, $S_{references}(X)$ and $S_{subobjects}(X)$, the final similarity value between any of those objects and object X, is obtained as: [1] Note that $t(X)$ is the type of a node X, and, e.g., $t(X).aweight$ is the weight for attributes for all nodes of the type $t(X)$.

$$S(X,Y) = \frac{(t(X).aweight * s_{attributes}(X,Y) + t(X).rweight * s_{references}(X,Y) + t(X).soweight * s_{subobjects}(X,Y)}{t(X).aweight + t(X).rweight + t(X).soweight}.$$

[1] This is true only in case that the object X does not have keys, in the case that the object X do have keys, the value of keys determine the similarity value, in a binary fashion

```
float attributeSimilarity(ComparisonMElement o1, ComparisonMElement o2,
        CalculationConfiguration c) {
    float similarities[] = new float[o1.attributes.length];
    for (i = 0; i < o1.attributes.length; i++) {
       MAttribute o1attribute = o1.attributes[i];
       MMAttribute attributetype = o1attribute.type;
       MAttribute o2attribute = o2.attributes.getByType(attributetype);
       SimilarityFunction sf = c.getSimilarityFunction(attributetype);
       similarities[i] = sf.calculate(o1attribute, o2attribute);
    }
    float result = 0.0f;
    if (c.getAttributesSimilarityExpression(o1.getType()) == null) {
        for (i = 0; i < o1.attributes.length; i++) {
            result = result + similarities[i];
        }
        result /= o1.attributes.length;
    } else {
        SimilarityExpression se = c.getAttributesSimilarityExpression(
                o1.getType());
        result = se.calculate(similarities[i]);
    }
    return result;
}
```

Listing 3.1: Pseudocode of the similarity function for attributes

The similarity function between two objects can now be defined as:

$$s(X,Y) = true \text{ iff } S(X,Y) > t(X).overallthreshold$$

The interpretation of the similarity function is as following: An object is similar to another object only if the similarity value between them is larger than a predefined threshold. The final similarity value between two objects is calculated by taking into account similarities between attributes, reference, and sub-objects. However, attributes, references and sub-objects contribute to the similarity value in different ways, denoted by weights. Thus, if attributes have a stronger influence on the similarity between objects of a specific type, the weights assigned to attributes should be larger than weights assigned to references and sub-objects.

Similarity functions for attributes, references, and sub-objects are given next. Listing 3.1 shows the simplified pseudocode of the similarity function for attributes. The actual implementation described in Appendix A uses results from the field of multidimensional search, and is more complex in order to be fast.

```
float referenceSimilarity(ComparisonMElement o1, ComparisonMElement o2,
        CalculationConfiguration c) {
    ReferenceSets rs1[] = o1.splitReferencesByType();
    ReferenceSets rs2[] = o2.splitReferencesByType();
    float similar = 0.0f;
    int totalsimilarReferences = 0;
    int totalReferences = 0;
    for (i = 0; i < rs1.length; i++) {
       int numberOfSimilarReferences = compareReferenceSets(rs1[i], rs2
           [i]);
       totalsimilarReferences += numberOfSimilarReferences;
       totalReferences += rs1.length;
    }
    similar = totalsimilarReferences / totalReferences;
    return similar;
}
```

Listing 3.2: Pseudocode of the similarity function for references

Next, we will consider the similarity function for references. As already mentioned, each reference is related to a certain *MMReference* element which we will call the *type* of the reference. We will separate the sets of reference instances of objects being compared into sub-sets based on the type of the reference. Next, we will only calculate the similarity between sets that contain references of the same type. The "naïve" solution of calculating the similarity between two references is to calculate the similarity between two objects that those references reference. However, the problem is that the referenced objects might not yet be processed (since the first step processes model trees bottom-up). Thus, in order to calculate reference similarities, the first step of the algorithm needs to be split into two sub-steps. First, similarities between objects need to be calculated without taking references into account. Second, the similarities found should be used to calculate reference similarities and to *fix* object similarities. This is also known as *similarity flooding*. However, unlike in traditional approach to similarity flooding, where the similarities are recalculated until they stabilize, we make a more pragmatical approach, where the number of *floods* can be pre-defined, thus increasing the speed of the algorithm. A simplified pseudocode of the similarity function for references is presented in Listing 3.2. The similarity function for sub-objects is very similar to the similarity function for references. However, since the similarities between sub-objects are known while calculating the similarity of their parent object, it is not necessary to have two sub-steps as with

```
float subobjectSimilarity(ComparisonMElement o1, ComparisonMElement o2,
        CalculationConfiguration c) {
    SubobjectSets sos1[] = o1.splitSubobjectsByType();
    SubobjectSets sos2[] = o2.splitSubobjectsByType();
    float similar = 0.0f;
    int totalsimilarSubobjects = 0;
    int totalSubobjects = 0;
    for (i = 0; i < sos1.length; i++) {
       int numberOfSimilarSubobjects = compareSubobjectSets(sos1[i],
           sos2[i]);
       totalsimilarSubobjects += numberOfSimilarSubobjects;
       totalSubobjects += sos1.length;
    }
    return similar;
}
```

Listing 3.3: Pseudocode of the similarity function for sub-objects

references. The pseudocode of the similarity function for sub-objects is given in Listing 3.3.

Bottom-up calculation of similarities

In the first step of the algorithm, model trees are traversed bottom-up (the root of the tree is the model itself). At each level of the tree, the objects are separated into groups by two criteria: their type, and their parent. Thus, groups are made of objects that have the same parent, and are of same type. This is done in both trees. Next, each group of objects in the first tree is compared to a group of objects in the second tree which has the same type, and the parent of the same type (one group can be compared to several other groups). The comparison is done by selecting, one by one, objects in all groups of the first model, and comparing them to all objects in the matching groups of objects from the second model. For each object in the first model, a list of similar objects in the second model is created. For each object in the created list a similarity between that object and the object in the first model is also retained.

The hash-value of an object is used to speed up the calculation of similarities between objects, because if the objects have the same hash-values, all of their sub-elements are considered to be identical. However, the use of hash-values is very limited, since it is computationally hard to calculate hash-values for objects

containing references.

During the calculation of similarities, if objects in a certain group have *Keys* (i.e. universally unique identifiers), the values of these *Keys* are used instead of a similarity function (i.e. objects are considered similar if they have the same key value). After the first step of the algorithm is done, for each object in model *A* there is a (possibly empty) list of similar objects in model *B*.

Top-down calculation of matchings

The second step consists of a recursive top-down traversal of the model trees. At each recursion step, a set of objects from one tree is compared to a set of objects from the second tree, and certain elements are possibly matched. At first step, the sets of objects are top level elements in both models. In subsequent steps, the sets of objects that are being compared are the sub-objects of the matched objects at the previous step. The recursion finishes when it is not possible to match any two objects in the two sets of compared objects (this includes the situation when one or both of these two sets are empty).

The matching itself is a combinatorial problem. The setup is as follows: there are two sets of objects, and each object in the first set has the list of objects it can be matched to in the second set (i.e. similar objects found in the first step of the algorithm). Furthermore, for all elements that an object can be matched to, a similarity value is also available. The goal is to maximize the number of matched elements based on these lists of possible matches and similarities. This is very similar to a known weighted bipartite graph matching problem - i.e. assignment problem. Thus, we use a modified Hungarian algorithm to find best matches of the sets of sub-objects for two matching objects. The modification is due to the fact that in an assignment problem the matchings are done to minimize the sum of weights. However, we would like to make matchings which maximize the similarities. However, since the similarities have an upper bound (value 1.0), we subtract the similarities from the value 1, and use this values as weights.

Note that the calculation algorithm presented in this chapter is the one that appeared in our paper presented in [111]. However, this algorithm does not calculate the moved elements (elements in model *A* that have a different parent in

model *B*). In order to calculate the moved elements, this algorithm needs to be extended with two more sub-steps, after the current step. In the first sub-step of the extended algorithm, similarities are calculated for all unmatched elements of the same type. In the second sub-step of the extended algorithm, all sufficiently similar elements of the same type are matched (as moved elements). We implemented this extension for our research in model co-evolution discussed in chapter 6.

Bottom-up calculation of differences

The third step consists of three sub-steps. The first sub-step calculates the differences in terms of deleted or changed sub-elements and attributes. The second sub-step determines added elements, and the third sub-step determines changes in references.

The first sub-step consist of a bottom-up traversal of a tree representing the first model in comparison. For a certain object it might be the case that this object has a match or that this object does not have a match. If an object does not have a match, it is instantiated as a *DeletedElement* difference object. The created difference model object is stored, and processed further when the parent of the deleted object is processed. It can be the case that the parent of this object was deleted also, in that case this difference object is deleted, because it is replaced by its parent difference delete object. Another case is that the parent of the deleted object was changed. In that case, the deleted difference object is included as part of the change object of its parent (an instance of the *ChangedElement* of its parent will contain this difference object). If an object has a match, then, if its matching object is completely identical, nothing happens. Otherwise, a *ChangedElement* difference object is created referencing the changed object, and for all attributes and sub-objects of this object change objects are calculated and stored in this difference object.

In the second sub-step, a top-down traversal of the tree representing the second model is performed, and all elements that are not matched are transformed into *AddedElement* difference objects.

In the third sub-step, all the references in the first tree are traversed, and for all

references, all elements that had their identifier changed (i.e. all objects that have been matched to objects such that the matching object identifier differs from the initial object identifier) have also their identifier changed in the references.

3.4 Conclusions

In this chapter we presented our extensions to state-of-the art approaches in the fields of presentation and calculation of model differences. Our approach to the presentation of model differences introduces an enhanced metametamodel, where both metamodels and models are directly instantiated from the metametamodel. This makes it possible to define a more fine-grained difference metamodel than with traditional metametamodels (e.g. MOF or Ecore). Furthermore, our difference metamodel is independent of metamodels, thus our approach is generic.

Our approach to the calculation of differences uses a highly configurable similarity function. This function can express four well-known types of matching: static-identity, signature-based, similarity-based, and language-specific. The algorithm exploits structural model properties and multidimensional search to obtain good precision and performance.

Chapter 4

Assessing the Quality of Tools for Model Comparison

In this chapter we discuss three research results that, combined, provide an experimental validation of our algorithm for the calculation of model differences described in the previous chapter. The first result discussed is a method for assessing the quality of model comparison tools. This method defines means of measuring and interpreting three quality attributes of a model comparison tool: speed, precision and recall. The next result discussed is a set of experimental data that we developed. This data set can be used with the described method to assess the quality of model comparison tools that support metamodel-independent model differences. Next, we describe two tools for metamodel-independent model comparison; one of which we developed based on our model differences calculation algorithm, and one commercial tool. As a final result, we describe a series of experiments that were performed by applying the defined data, according to the

defined method, to the two described tools. The results of the experiments show that the tool we developed is slightly slower than the commercial tool, but that it calculates the model differences more accurately.

4.1 Introduction

Advances in model driven software engineering have paved the way for many new research areas, one of which is *model comparison* - a process of comparing two models to obtain the difference between those models. This process is essential for understanding the evolution of a model over a period of time, since this involves comparing different versions of that model. In existing approaches for model comparison [50, 79, 85, 111], the difference between models is represented as a *set of atomic differences* that represent added, deleted, or changed model elements.

4.1.1 Comparing Models

The process of comparing models consists of the two subprocesses: *model-matching* and *difference calculation*. The model-matching subprocess concerns the discovery of relations between model elements that *model the same entity* in the models being compared. There are four recognized types of model matching: static-identity based, signature based, similarity based and language specific [83]. Each of these approaches leverages different knowledge on the syntax and semantics of matched models, in order to provide as accurate matching as possible. Once the model elements are matched, a differences-calculation subprocess uses the discovered matchings, and returns the difference between two models.

Model comparison can be metamodel-dependent, or metamodel-independent. In metamodel-dependent model comparison, the comparison process is tailored towards comparing models that conform to a specific metamodel (e.g. UML). In this regard, both the syntax and semantics of the models conforming to that metamodel are taken into consideration during the comparison [123].

In metamodel-independent model comparison, the comparison process is not tailored to a specific metamodel, but can be used to compare models regardless of their metamodel. However, although the existing approaches to metamodel-independent model comparison use a generic model-matching algorithm, most of them offer some way of adapting the comparison process to suit a particular metamodel. For example, model comparison processes that use similarity-based model matching can be configured by using weights [79, 111].

The advantage of metamodel-independent approaches to model comparison is their generality. The disadvantage is that the completely automated part of the comparison process is based only on the syntax of the models being compared, disregarding any (dynamic) semantics of models. Thus, metamodel-independent approaches usually provide less accurate results than metamodel-dependent model comparison approaches. For example, two UML state-machines may be syntactically different, but can exhibit the same behavior. Thus, if only the syntax of state-machines is taken into consideration, an incorrect or misleading difference could be reported.

4.1.2 Contributions

In both metamodel-dependent and metamodel-independent approaches to model comparison, it is required to assess the quality of a particular approach. The reason for this requirement is that there is no perfect approach, i.e., that would be able to calculate all the correct differences and only the correct differences. One way of assessing the quality of a particular approach is through assessing the quality of a tool implementing that approach. In this chapter, we define a method and an benchmarking data set, that can be used to assess the quality of model comparison tools through controlled experiments.

In our approach, the quality of the tool is assessed by measuring three metrics on the defined benchmarking data. This list of three measured metrics can be extended easily. In particular, we choose to measure comparison *speed*, *precision*, and *recall*, which are metrics commonly used for measuring the quality of tools for comparing models or metamodels (see e.g. [63, 123]). These metrics were chosen because they reflect the most desirable properties from the user

point of view: the comparison tool should provide results fast[1], it should return as much correct differences as possible (i.e. it should have a high recall), and it should return as few incorrect differences as possible (i.e. it should have a high precision).

In this work, we focus on metamodel-independent approaches to model comparison, though our method and the benchmarking data can also be applied to metamodel-dependent approaches. We chose this approach since, in our opinion, metamodel-independent approaches are a preferred solution for model comparison. The reason for this is that metamodel-independent approaches can be used off-the-shelf to compare models conforming to a variety of metamodels, while metamodel-dependent approaches must be developed for each metamodel separately. Thus, a company utilizing many metamodels or with evolving metamodels will prefer a configurable off-the-shelf metamodel-independent tool, rather than multiple metamodel-dependent tools which would require a model-comparison expert to develop and maintain.

We applied our method, by using the defined benchmarking data, to two model comparison tools that offer metamodel-independent model comparison, namely EMFCompare [9] and RCVDiff [113]. EMFCompare is a tool used in the Eclipse Modeling Framework (EMF) for both model comparison and model merging of models that are instances of Ecore [11]. RCVDiff is an academic tool that can be used for comparison and visualization of models conforming to our enhanced metametamodel (EMMM) described in Chapter 2. These particular tools were selected since both of them are publicly available, are past the initial development phase, and are sufficiently documented. We have also considered Fujaba [15] (which utilizes SiDiff [79] comparison algorithm), and OdysseyVCS [99]. However, these tools are focusing on comparison of UML models (i.e. they are metamodel-dependent) and thus did not fit our requirements.

The rest of this chapter is organized as follows: In Section 4.2, our method for assessing the quality of model comparison tools is described in detail. Thereafter, in Section 4.3, a benchmarking data set is described. Next, in Section 4.4, a brief description of RCVDiff and EMFCompare is given, whereafter we present and

[1] *Fast* is a relative term, and it depends both on the modeled system, and on the developers that are developing the system.

discuss the results of assessing the quality of those tools by using our defined method and our benchmarking data set. Finally, in Section 4.5, we discuss possible improvements to our method and benchmarking data set, and we offer some concluding remarks.

4.2 Method for assessing the quality of model comparison tools

Our method assumes that the compared models conform to a metamodel that conforms to Ecore [11] (or that the compared models (and the metamodels that they conform to), can be transformed to models (and metamodels) conforming to Ecore). We have chosen Ecore as the metametamodel (and not MOF [20] or EMMM [111]) because it is both simple and widely used.

To measure the quality of a model-comparison tool using our method, the tool should be able to return the *model of difference* between two models. Moreover, it is required that the returned difference model consists of atomic differences which represent added, deleted, or changed model elements. These (atomic) differences are used to calculate the precision and recall. If the models being compared are denoted as the *original* and the *target*, then:

- A *deleted* model element is a model element that exists in the original model but there is no model element in the target model that models the same entity.
- An *added* model element is a model element that exists in the target model but there is no model element in the original model that models the same entity.
- A *changed* model element is a model element in the original model that models the same entity as a model element in the target model, but that is not identical to the corresponding model element in the target model.

In this paper, the *changes* to a model element are constrained to changed values of attributes, and changed, added, or deleted reference instances. Added or deleted sub-elements are considered as added or deleted model elements, and

changed sub-elements are considered as changed model elements. Thus, if a model element *A* contains a model element *B* that is changed, then, although element *A* is indirectly changed, it is not considered as a changed element in this method.

As already mentioned, we have chosen to measure three metrics commonly used for model and metamodel differences: comparison speed, precision, and recall. Comparison speed is an efficiency metric, while precision and recall are functionality metrics [75].

Comparison speed is measured as the time (in milliseconds) elapsed during the differences calculation process. In order to describe the precision and recall metrics, several other terms must be introduced.

- *correct_differences* is the number of atomic differences between two models as specified by the designer that evolved the original model. In case that compared models are not evolutionary related, an oracle must exist which is able to provide the correct differences (i.e. the size of the set of correct differences);
- *found_differences* is the number of atomic differences between two models as returned by the comparison tool (i.e. the size of the set of returned differences);
- *matching_differences* is the number of atomic differences returned by the comparison tool which are in the set of correct differences (i.e. the size of the intersection of the sets of correct and returned differences).

Precision is defined as the measure of correctness or fidelity; higher precision of a tool means that less incorrect differences are returned by the tool. Precision is a rational value between 0 and 1 and is calculated as:

$$precision = \begin{cases} \frac{matching\_differences}{found\_differences} & \text{if } found\_differences \neq 0 \\ 1 & \text{if } found\_differences = 0 \\ & \text{and } matching\_differences = 0 \end{cases}$$

Recall is the measure of completeness of comparison; higher recall of a tool means that the tool finds more correct differences. Recall is a rational value

between 0 and 1 and is calculated as:

$$recall = \begin{cases} \frac{matching\_differences}{correct\_differences} & \text{if } correct\_differences \neq 0 \\ 1 & \text{if } correct\_differences = 0 \\ & \text{and } matching\_differences = 0 \end{cases}$$

It is clear that in order to calculate the precision and recall metrics, it is required to have a set of *correct atomic differences* for each pair of models being compared. Thus, the set of samples for assessment experiments should consist of triples of models:

$$experiment\_samples = \{(M_1, M_2, DM)\}$$

The first model in each sample triple (M_1) is considered as the original model. The second model in each triple (M_2) is considered as the evolved (version of the) original model (this model will be referred to as *the target model*). The last model in each triple (DM) is (the model of) the *correct difference* between the original and the target model, i.e. $DM = diff(M_2, M_1)$. For example, if the name of a class was changed from A to B by a model designer, a *correct difference* model would include only the *atomic* difference which reports that the class has changed its name. An example incorrect difference would contain two atomic differences, one representing the deletion of a class named A, and another representing the addition of a class named B. In order to use a sample as a benchmark sample, each sample should be accompanied with the benchmark result[2]:

$$benchmark\_result = (precision, recall, speed)$$

Measurement of the selected metrics by a tool for model comparison is done by the following procedure, for all experiment samples: First, an experiment sample (M_1, M_2, DM) is selected. Next, by using the tool, models M_1 and M_2 are compared. The difference model returned by the tool is denoted as $ReturnedDM$. Next, one should calculate the *correct_differences* and the *found_differences* numbers. The *correct_differences* is the number of elements in the DM difference model, and *found_differences* is the number of elements in the $ReturnedDM$ difference model. Next, DM and $ReturnedDM$ must be compared in order to calculate the *matching_differences* number. In order to compare DM and $ReturnedDM$,

[2]The benchmark result is used to compare the result returned for the assessed tool with the result of a *benchmark* tool.

it should be possible to relate a particular atomic difference element to a particular model element. In order to calculate the *matching_differences* number, the following algorithm is applied:

- Find all pairs of deleted elements in *DM* and in *ReturnedDM*, that are related to the same model element.
- Find all pairs of added elements in *DM* and in *ReturnedDM*, that are related to the same model element.
- Find all pairs of changed elements in *DM* and in *ReturnedDM*, that are related to the same model element, and that their changes are identical.
- Combine all the found elements in a set, and calculate the size of that set. This size is the *matching_differences* number.

Thereafter, the values of precision and recall for the experiment sample can be calculated (and all three calculated metrics can be compared to the benchmark results for the used sample).

4.3 Data sets for assessment experiments

In our assessment experiments, we used two subsets of model triples. The first of those subsets is a small set of manually defined CIF [5] model triples. This set of model triples is described in the first subsection of this Section; it represents models actually encountered *in practice*. The second subset of model triples contains a large number of models *automatically generated* by several generator tools we developed. The generation process, as well as the tools used to generate the triples in this test set, are described in the second subsection of this Section.

We chose to use two experiment sets because we did not have access to a large number of manually defined model triples. Thus, in order to increase the confidence in the correctness of the measured results, we decided to generate another, larger, set of model triples. Also, it is easier to control certain model characteristics, such as size, complexity and coverage of various features in a generated set.

4.3.1 Manually defined data set

CIF stands for *Compositional Interchange Format*. It has been designed as a generic interchange format for hybrid transition systems [5, 43]. CIF models can be created in the Eclipse framework and have two possible representations: grammar-based by using Xtext [34] and model-based by using Ecore [11].

For this study, we selected several CIF models developed by designers in the Department of Mechanical Engineering at Eindhoven University of Technology. All selected models are versioned and can be found in the CIF repository. However, since most of the available CIF models are in the grammar-based representation, model transformations were used to obtain the model-based representation of the selected models. Each selected CIF model gave rise to one experiment sample triple. The first component of each of the CIF model triples is the first version of a selected model in the repository. The last committed version in the repository is taken as the second component of each triple (the revision number of each version of each selected model is mentioned in the results). The third component of each triple, i.e., the correct differences model between an original model and an evolved model, was obtained with the help of developers that evolved the selected models. In particular, the developers were given a tool for manual model matching, that allowed them to manually match the elements in both the original and the evolved model. Based on the manual matchings, the tool calculated the correct differences between those models. The calculated differences are represented by a differences model conforming to the differences metamodel depicted in Figure 4.4. After the calculated differences were inspected to validate their correctness, they were used as a third component in a sample triple.

4.3.2 Generated data set

The second set of experiment sample triples contains generated models. The models constituting the first component of each triple are Ecore-based, and are generated by a model-generator tool. Moreover, the metamodels of those models are generated by a metamodel-generator tool. These generator tools are Java-based, and can be configured to produce metamodels or models having a specified (but possibly randomized) structure. A detailed description of the generation

process of the first component of each of the generated sample triples is given in Section 4.3.2. The second and third component of each triple are also automatically generated. A detailed description of the generation process of the second and the third components is given in Section 4.3.2.

Our approach to the generation of metamodels and models is similar to the approach presented in [97]. However, while the authors of [97] generate only models, we generate both models and metamodels. Moreover, our generation process can be configured by using several random distributions, and not only the uniform one proposed by the authors of [97]. Also, while it is possible to use a transformation language such as Epsilon [13] or ATL [1] to generate metamodels and models, we have opted to use Java for this purpose. The reason is that our generation process relies heavily on several random distributions which are not included in transformation languages.

Generating models and metamodels

The metamodel-generator tool is configured by using configuration models conforming to the (metamodel-)configuration metamodel depicted in Figure 4.1.

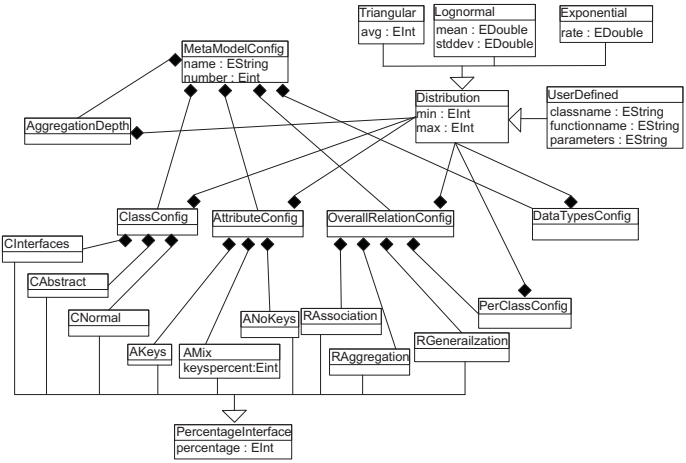

Figure 4.1: Metamodel of the configurations used by metamodel generator

Each metamodel-generator configuration contains an instance of a *Metamodel-Configuration* class. Values of attributes of instances of this class specify the number of metamodels that should be generated, the names of metamodels to be generated, and the location in which the generated metamodels should be stored. Each *MetamodelConfiguration* instance can contain sub-objects. A *ClassConfig* object specifies how the classes should be created. In particular, one can specify the percentage of normal, abstract or interface classes. Moreover, one can specify the distribution which is used to calculate the number of generated classes. An *OverallRelationConfig* object specifies the overall number and type of references generated, and *PerClassConfig* object specifies the number and type of references generated per class [3]. An *AttributeConfig* object specifies the number and type of attributes generated per class. A *DataTypesConfig* object specifies the number of data types generated. In the version of the metamodel-generator used for this study, we have simplified the generation such that each generated metamodel has only one package, and that the only data type used is *EString*.

In order to configure the metamodel-generator tool, we first obtained a large set of metamodels from the ATLAN metamodel zoo [2]. Next, we measured parameters relevant to a metamodel-generator configuration on the obtained set of metamodels (e.g. the distribution of classes in metamodels, or the distribution of attributes in classes), and we set-up one metamodel-generator configuration by using those parameters. It is important to note that in both the metamodel-generator tool and the model-generator tool, a random distribution must be defined for each element in the configuration. This distribution is used to randomize the structure of the generated metamodels and models. We obtained the distributions and their parameters, by fitting the measured data using the tool CumFreq [7]. For this study, we have generated 10 metamodels.

The model-generator tool is configured by using configuration models conforming to the (model-)configuration metamodel depicted in Figure 4.2. The model-generator configuration specifies the number of created models, the name of models, the tree-structure of models, and the structure of the objects constituting the model. It is important to note that our model-generator tool considers only syntactical features of metamodels in order to generate models. Thus, semantical

[3] In case of conflicts, the values generated by using the *OverallRelationConfig* object have priority.

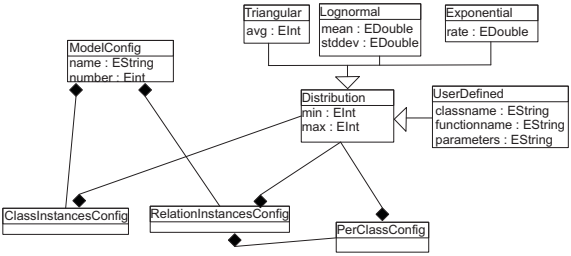

Figure 4.2: Metamodel of the configurations used by model generator

constrains (e.g., OCL constraints) are not taken into consideration. This is not an issue if metamodel-independent model comparison tools are used, since the comparison algorithms are based only on the syntax of models, but could be an issue if a larger data set should be generated for testing model-dependent model comparison tools.

To configure the model-generator tool we have used a large set of ATL models found in the ATL transformations zoo [3]. We measured the parameters relevant to the model-generator configuration (e.g., we measured the number of objects per model, and we used the CumFreq tool for the distribution fitting of this parameter), and we set-up one model-generator configuration by using those parameters. For the purpose of this validation study we have generated 50 models for each generated metamodel, hence 500 models in total.

Generated differences and patching models

To automatically generate the correct differences for models generated by the model-generator tool, we have created a tool for generating model differences. To generate the evolved versions of the models generated by the model-generator tool, we have created a tool for applying the generated differences. Since the role of the model differences in this case was to ”mutate” the initial models, in this section the term *model mutations* is used as a synonym for the term *model differences*. Also, the term *patching* is used to denote the application of a model difference to an original model in order to obtain the evolved (i.e. target) model.

The tool for creating model mutations, similarly to the other two generator tools,

can be configured by using configurations that are models conforming to the (mutator-)metamodel depicted in Figure 4.3.

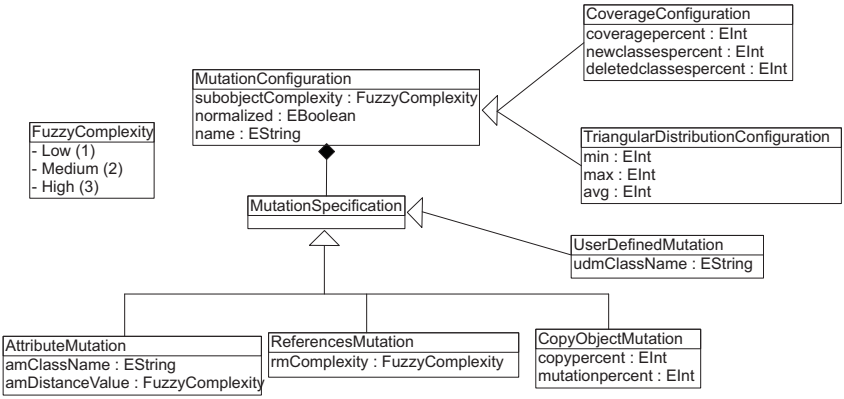

Figure 4.3: Metamodel of the configurations used by model mutator

There are two types of configurations for the model-mutator tool. The first type are the *distribution* configurations, that are instances of *TriangularDistribution-Configuration* class. The second type are *coverage* configurations, that are instances of *CoverageConfiguration* class.

Distribution configurations are used to generate a random number of mutations by using a predefined (triangular) distribution. The parameters of the distribution (minimum, maximum and an average number of generated mutations) are set as the attributes of the configuration object. *Coverage* configurations are used to generate mutations that affect a certain percent (i.e. a certain coverage) of all elements in the model. The configurations of the *Distribution* type can be used to generate a predefined number of mutations, while the configurations of the *Coverage* type can be used to generate mutations which affect a certain percentage of all model elements in a model.

For both configuration types, mutation aspects of attributes, references, or subobjects of elements in models can be specified. For attributes, the name of the function that will do the actual mutation, and (by using fuzzy numbers) the distance between the initial value and the evolved value of the attribute, must be specified. For references, the complexity of the references (also by using fuzzy

numbers) must be specified. In this study, the complexity of the references is reflected in the *connectedness* of the mutated references. Thus, more complex references mutations are those that result in more connected references. For sub-objects, the percentage of objects that will be copied, and the percentage of mutations applied to those copies must be specified.

For this validation study, we have developed 10 configurations. Eight of those are coverage configurations, that combine different coverage percentages and different mutation complexities. Two of those are distribution mutations, with the same distribution but different mutation complexities for attributes and references.

The result of the model mutator tool is a set of operation-based model differences that conforms to the metamodel depicted in Figure 4.4.

This set of difference operations is designed based on the set of difference operations described by Herrmannsdoerfer et al. [72]. However their differences operations are not concrete enough, and cannot be immediately used as a patch. Thus, we improved this metamodel such that its instances can be immediately used to generate the evolved model from the original model (i.e., it can be used to *patch* the original model). Notice that the differences metamodel used for mutations differs from the differences metamodels used in RCVDiff and EMFCompare. The reasons for adopting another differences metamodel are of a practical nature: The differences metamodel used by RCVDiff is not appropriate since the models and mutations should be Ecore-based. The differences metamodel used by EMFCompare is not suitable to be used as a patch (one of the main reasons for this is that the persisted difference model contains neither original nor the changed values of attributes.).

Moreover, we have used the *operation-based* differences metamodel instead of a *state-based* differences metamodel (used by RCVDiff and EMFCompare) for two reasons. The first reason is that the operations used in this metamodel are easily comparable to atomic differences used by RCVDiff and EMFCompare. Thus, it is easy to calculate the *matching_differences* number. The second reason is that the operations can be easily used as a *patch*. Thus, the creation of a *model-patcher tool* is simplified.

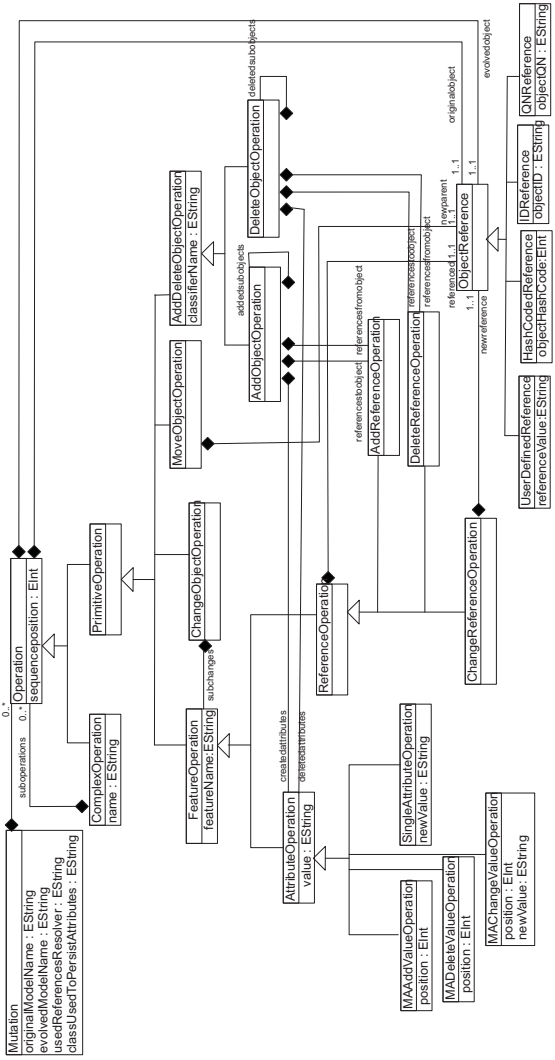

Figure 4.4: Metamodel of the operation based differences produced by model mutator

Based on the differences metamodel depicted in Figure 4.4, we developed a *model-patcher tool* that takes the model and a mutation and produces a mutated (i.e. evolved) model. By using the model-mutator tool and the model-patcher

tool, we have obtained the second and the third component of all triples in the second test set.

4.4 A comparative study of EMFCompare and RCVDiff

In this section, we first briefly describe the RCVDiff and EMFCompare tools for comparing models. Thereafter, we present the results obtained from those two tools by applying our method described in Section 4.2, and by using the benchmarking data sets described in Section 4.3. Finally, we discuss the assessment results[4].

4.4.1 RCVDiff

RCVDiff is a tool that can be used for the calculation and visualization of model differences [113]. The metamodel of differences returned by RCVDiff is depicted in Figure 4.5. In this study, moved elements were not considered.

RCVDiff uses a similarity-based model matching algorithm that can be configured to support all other types of model matching (i.e., static-identity based, signature based, and language-specific). A matching algorithm used by RCVDiff is configured by an instance of a configuration metamodel depicted in Figure 4.6. Each configuration (i.e., an instance of a configuration metamodel) is defined for one metamodel, and can be used to improve the matching results when instances of that metamodel are compared. In particular, for each metamodel element, one can instantiate a configuration element that specifies how to match the instances of that metamodel element. It is possible to specify the influence of attributes, references and sub-elements on the comparison process. Moreover, it is possible to specify a user defined function that will be used instead of the predefined algorithm to compare the instances of a particular metamodel element. This function can be used to take into account both the syntactic and the semantic aspects of model elements during the comparison. Details of the configuration metamodel,

[4]The experimentation data and all results can be found at [4].

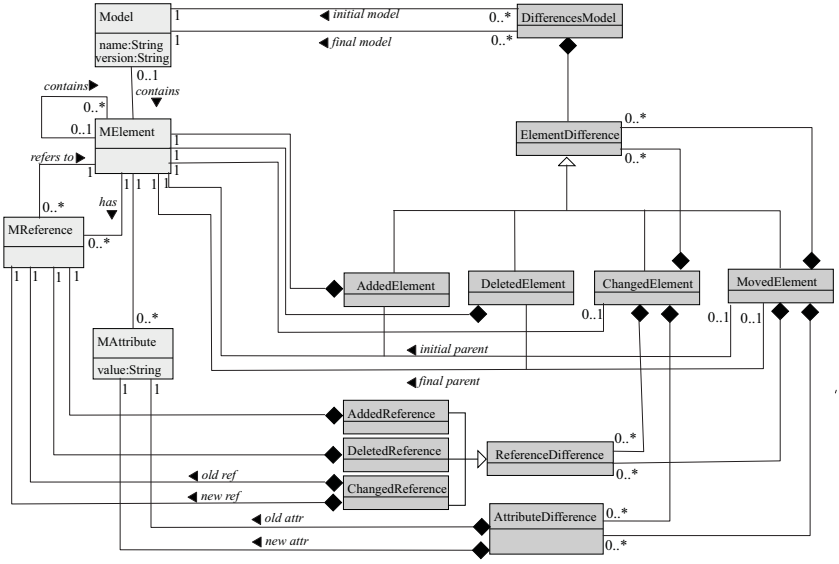

Figure 4.5: RCVDiff differences metamodel

and the matching algorithm of RCVDiff can be found in Section 3.3.3.

4.4.2 EMFCompare

EMFCompare is a tool that can be used for *2-way* and *3-way* comparison of Ecore-based models, as well as for manual or automatic merging of Ecore-based models [9]. EMFCompare is an integral part of the Eclipse framework starting from the version *Helios* [10]. The comparison process in EMFCompare is split into two sub-processes: model matching and differences calculation. EMFCompare uses a generic, similarity-based, heuristic model matching algorithm, and offers the possibility of manually defining custom matching algorithms. After the models have been matched, the calculation of differences is done by a generic difference algorithm, that is (almost) trivial and does not warrant user extensions.

The differences returned by EMFCompare conform to a metamodel for state-

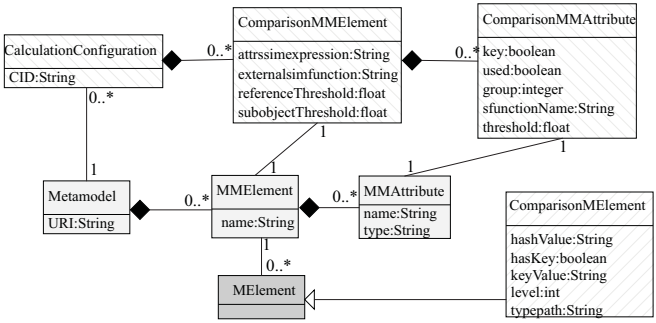

Figure 4.6: RCVDiff configuration metamodel

based differences depicted in Figure 4.7.

Unlike RCVDiff, that uses a declarative approach to model comparison, EMFCompare uses an imperative approach, and cannot be configured easily. In particular, the matching algorithm used by EMFCompare is hard-coded in Java classes, and there is no clear description on how to adapt the algorithm for a specific metamodel. For example, a source of the class named *StatisticBasedSimilarityChecker* contains a set of weights that are used in comparison. However, those weights are not related to a specific metamodel element, but are used for all metamodel elements. Thus, extending EMFCompare is a complex and tedious process, which requires extensive knowledge of Java, Ecore, and EMFCompare itself.

4.4.3 Results

Manually defined data set

The measurement results, in case that CIF model-triples set is used, are given in the Table 4.1 [5].

In this case, we performed three experiments. In the first experiment, we used

[5] Next to the model name, revision numbers of compared versions of a particular model are given.

			RCVDiff			RCVDiff$_{configured}$			EMFCompare		
Model name	Model Size	Mutation Size	Recall	Precision	Performance	Recall	Precision	Performance	Recall	Precision	Performance
bottle (r1348,r1550)	422	57	0.54	0.28	814	0.54	0.30	585	0.61	0.74	250
buffer (r1374,r1571)	212	20	0.87	0.50	814	0.9	0.53	295	0.9	0.78	47
parallel composition (r1514,r1732)	285	25	0.12	0.04	1159	0.12	0.04	729	0.04	0.02	94
aut_def2 (r1415,r1431)	164	16	1	0.13	762	1	0.13	568	0.81	0.76	62
aut_def2_roundtrip (r1430,r1557)	400	11	1	0.05	982	1	0.05	927	0.36	0.25	156
auts (r1415,r1557)	531	25	0.16	0.03	783	0.88	0.59	742	1	0.86	125
scoping_core (r1431,r1557)	39	3	1	1	273	1	1	167	1	1	31
Averages	293	22	0.68	0.29	744.57	0.77	0.38	573.28	0.67	0.63	109.29

Table 4.1: Measurements results, for a set of manually defined models

our method with the defined data set, to measure the defined metrics on RCVDiff, with the default values of configuration parameters. In the second experiment we configured RCVDiff to work with CIF models, and measured the metrics again. In the third experiment we measured the defined metrics on EMFCompare, by using the default EMFCompare configuration.

It is clear that EMFCompare is an order of magnitude faster than RCVDiff (however, almost all comparison times are less then 1 second, and are thus acceptable to a user). The reason for this is a complex metamodel of CIF models (having around 60 elements and many relations), which is transformed in a very large EMMM metamodel (around 1200 elements), slowing down the comparisons in RCVDiff (it takes around 160 milliseconds to load and process the CIF metamodel and its default configuration). However, while the recall values of RCVDiff in the first experiment are practically equal to the ones obtained by the third experiment, the recall values obtained by RCVDiff in the second experiment are higher than the ones obtained in the first and third experiments. This was expected since RCVDiff was configured in this case. The precision of RCVDiff is considerably lower than the precision of EMFCompare in both cases. In order to find the reason for this behavior further research is required.

Generated data set

Since in the case of generated models, the tools were compared by using 500 models, but only 10 different mutation configurations, the results are shown as averages for each mutation configuration. Each mutation configuration is applied to 5 models conforming to each of 10 metamodels, thus the average results are over 50 models. The results, sorted by a configuration number, are given in

Table 4.2 [6].

			RCVDiff			EMFCompare		
Configuration ID	Avg. Model Size	Avg. Mutation Size	Avg. Precision	Avg. Recall	Avg. Performance	Avg. Precision	Avg. Recall	Avg. Performance
C1	242.44	206.54	0.78	0.59	296.06	0.46	0.34	95.52
C2	186.72	267.8	0.63	0.67	153.38	0.46	0.55	106.42
C3	207.46	213.38	0.74	0.62	196	0.48	0.36	109.16
C4	239.36	406.86	0.66	0.67	236.74	0.49	0.55	167.62
C5	198.68	191.62	0.74	0.59	157.04	0.47	0.34	92.9
C6	202.6	333.26	0.64	0.71	202.7	0.47	0.54	132.32
C7	199.52	189.92	0.76	0.56	196.42	0.48	0.39	93.62
C8	215.76	322.46	0.67	0.69	193	0.47	0.65	122.06
C9	230.44	198.26	0.74	0.57	200.3	0.48	0.35	112.4
C10	199.84	432.1	0.62	0.64	186.36	0.44	0.48	115.22
Column averages	212.28	276.22	0.7	0.63	201.8	0.47	0.45	114.72

Table 4.2: Measurements results, for a set of automatically generated models

In the case of generated models, the speed of RCVDiff and EMFCompare is in the same order of magnitude. However, precision and recall values obtained by using RCVDiff, are, in average, significantly better than those obtained by using EMFCompare.

4.4.4 Threats to validity

In this section we will discuss threats to the validity of our experiments. The first threat is concerned with the requirement that the experimental data must contain *correct differences*. In our manually defined experimental data set, this threat was addressed by having the users, that developed and evolved the selected models, define the correct differences. The users were given a tool for this purpose, and, based on the users mappings, this tool produced both a differences model, and a human-readable text notation that could be checked for correctness of the calculated differences. In our generated experimental data set, the correct differences are generated, and are thus correct by construction.

The second threat is concerned with the soundness and completeness of the experimental data set. A sound experimental data set is a correct one. A complete data set is the one that contains all types of atomic model differences that can be calculated by the tools.

The models in both experimental data sets are sound - all models are Ecore-

[6] All the columns except the configuration IDs contain averages

based, and can be loaded by Eclipse. The correct differences in the manually defined experimental data set are sound since we checked all the differences manually. The correct differences in the generated experimental data set are sound by construction. The choice of operation-based differences makes this easy to prove, since by generating correct operations, one also generates correct differences.

Since our manually defined data set was relatively small and we could not guarantee its completeness, we decided to generate a larger data set to ensure completeness. However, we did not formally check the experimental data set for completeness.

Another threat to validity emerges from the fact that we used a different difference metamodel in the data sets, then difference metamodels used by the tools. In order to address this threat, we first labeled all elements in all models. Next, we related each atomic difference (in all difference models) to a particular model element, by using the labels of those model elements. Then, we defined a mapping between the correct difference models used in the data sets, and the difference models returned by the tools. Next, for each atomic model difference in the data set, we checked if there is an atomic model difference returned by the tool, that is related to the same model element. If this was the case, and there was a mapping between these differences, we considered these differences as the same.

4.4.5 Discussion

The results show that the speed of the comparison, of the two compared tools, is in the same order of magnitude if simpler metamodels are used, and it is in favor of EMFCompare when complex metamodels are used. Nevertheless, both tools calculate the differences in a timespan acceptable to the user (up to a few seconds).

In the case of manually defined experiment data, if the tools are not configured, the average recall is almost the same for both tools, and EMFCompare has a better precision. However, after configuring RCVDiff, both its precision and recall increased.

In the case of generated experiment data, the recall and precision values for EMFCompare tool are around 0.5, which means that in average half of the returned differences are correct. The average precision and recall of RCVDiff in this case are 0.7 and 0.63. However, the values of 0.5, or even 0.7, for precision and recall are surprisingly low, and further research is required to discover reasons for this behavior.

4.5 Conclusions and Future Work

In this chapter, we have described a method and a benchmarking data set, for assessing the quality of model-comparison tools. Moreover, by using our defined method and benchmarking data, we have performed a series of experiments on the model-comparison tools EMFCompare and RCVDiff. Comparison of the results of those experiments show that the quality of these two tools is similar. Furthermore, the obtained results, together with the data set, can be used as a benchmark for assessing the quality of future tools for (metamodel-independent) model-comparison.

Future work could involve, for example, extending our method with new metrics. Also, further test cases could be included in the benchmarking data set; in particular, more manually defined test samples are needed. Moreover, we considered only metamodel-independent approaches and tools. Thus, while our method is applicable to both metamodel-independent and metamodel-dependent approaches, the data set is not applicable to metamodel-dependent approaches, and metamodel-dependant data could be included in the data set.

Next, the generator tools that were used to generate test samples, could be improved by considering even more structural aspects of metamodels and models (or even by considering semantical aspects of models). Moreover, these tools are also useful in other areas of Model-Driven Software Engineering, for example for generating larger test cases for other modeling facilities (e.g. for testing model transformations).

Furthermore, we did not take into account the comparison of models in which model elements are *moved* (a *moved* model element is an element in a target

model, that models the same entity as a matching model element in an original model, but the parents of a *moved* element and the matching element do not model the same entity in the compared models), and this could also be included in future extensions of the method and the benchmarking data set.

Finally, although we are confident that our experimental data set is complete, a formal proof of completeness of the experimental data set should be provided.

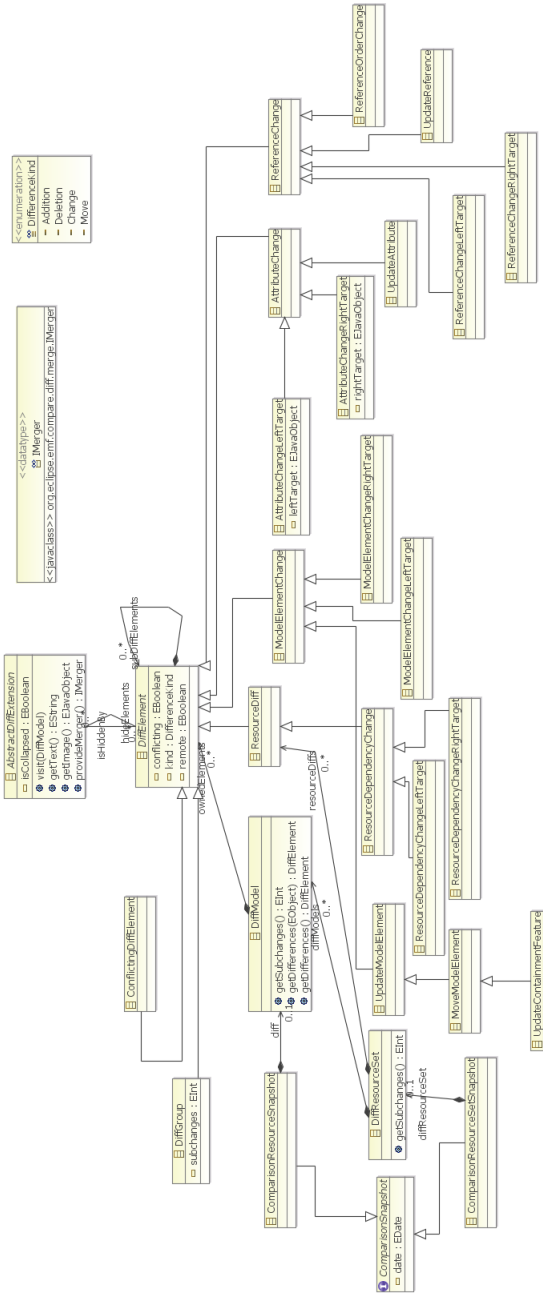

Figure 4.7: EMFCompare differences metamodel

Chapter 5

Model Differences Visualization

This chapter describes our approach to visualization of state-based model differences, that improves state-of-the-art approaches in this field. Our improvement is based on the fact that all existing approaches focus on only one visualization formalism in order to visualize model differences. We argue that in order to visualize model differences, multiple formalisms should be combined in order to achieve the best results. Thus, in our approach we combine two existing formalisms. The combination offers better understandability of the syntax and semantics of the model differences than each of its components in isolation. We validated our approach by building a tool for the visualization of model differences. The details of this tool are also described in this chapter.

5.1 Introduction

Model Driven Software Engineering (MDSE) is a field of Software Engineering which focuses on models as main design artifacts, and uses model transformations as means of relating models. Consequently, mature model configuration management systems are required to manage the complexity of modeled systems in MDSE environments. One of the major functions of model configuration management systems is model comparison. Model comparison (model differencing) is a complex process which consists of three concerns: representation, calculation, and processing of differences [84]. The rationale behind this separation of concerns is that usually it is not only required to calculate differences, but it is required to store, process, and visualize them in the context of a model configuration management system.

In this chapter we consider visualization of model differences. In [120] it was observed that traditional difference visualization approaches using text-based, tree-based or even diagrammatic visualization techniques, poorly scale with the size of the differences model. There are two reason for this behavior. The first reason is based on the fact that model differences are considered information content. Thus, we believe that their visualization should be based on the following idea of information visualization specified by Shneiderman [105]: overview first, zoom and filter, then details-on-demand. However, the existing approaches are not suited to visualize model differences in a way that completely supports the above mentioned idea. The second reason can be derived implicitly from the first reason: the traditional approaches use only one technique. However, in order to provide the best insight into the meaning of differences, more than one technique should be used. In Section 5.2 we give a detailed discussion on these two reasons.

In order to improve the model differences visualization capabilities provided by traditional approaches, we extend and combine two existing techniques: polymetric views [87] and a generic visualization framework for metamodel-based languages. We choose the first approach because it was already used in the context of visualizing model differences, and it provides good overview, zoom, and filtering capabilities. We choose to develop a generic visualization framework for metamodel-based languages in order to support the semantically rich

detailed representations of differences. A recent approach, described in [102], which combines a tree-based visualization technique provided by *EMF Compare* [9] with a visualization framework called *GMF* [18], uses similar ideas and makes a step beyond traditional approaches, but is tightly coupled with the Eclipse framework [10]. Our approach, and an associated tool, are generic, and both are metamodel and framework independent. The preliminaries required for understanding our approach are given in Section 5.3. Thereafter, in Section 5.4, a detailed description of our approach is presented. Next, in Section 5.5, details of a prototype tool that implements the described approach are presented. Finally, in Section 5.6 we discuss the results and propose some ideas for further research.

The main contributions of this chapter are:

- We discuss two reasons why traditional differences approaches poorly scale with the size of the difference models.
- We specify a new visualization approach that combines polymetric views and a framework for the visualization of metamodel-based languages, and thus scales excellently in the presence of large difference models.
- We present the details of a model difference visualization tool based on our approach.

5.2 Model Differences as Information Content

In our approach to visualizing model differences, we adopt the idea of information visualization proposed by Shneiderman [105]: Overview first, zoom and filter, then details-on-demand. The reason for adopting this idea is based on the fact that model differences are information that needs to be visualized. Thus, it is required to have overview capabilities, such that the global meaning of differences can be comprehended. Next, it should be possible to zoom in and filter differences, such that the user of configuration management systems (referred to as user in the rest of this section) can syntactically and semantically associate the differences to the parts of the models that those differences are related to. Finally, the selected differences should be rendered by using a sufficient level of detail to provide the best insight into their meaning.

Traditional approaches to difference visualization do not fully agree with Shneiderman's idea. It was observed that they scale poorly with the size of the difference model [120]. One of the reasons for this is that traditional visualization techniques cannot fully support Shneiderman's idea. In order to support this claim, we will provide a formalized definition of overview, zoom, filtering, and details-on-demand in the context of model differences. Thereafter, we will show why the traditional approaches have problems in satisfying these requirements. The important thing to know is that the ultimate goal of all the described concepts is providing the *maximum insight into the meaning of model differences*.

Useful overview techniques in the context of model differences should provide an overview of one or both models used to calculate the differences, and should relate the elements of those models to the elements in the differences model. Thus, an overview should allow the user to get a grip on the meaning of model differences in the context of the models that they are calculated from. The zoom should allow the users to focus on the interesting details. The filtering should allow the user to quickly navigate to the parts of the difference model that are important to her. The filtering should be based on the meaning of the differences, not only on their structure. The details-on-demand should allow the user to extract and focus on the details of selected differences.

Having defined the concepts of overview, zoom, filtering, and details-on-demand, we will provide an explanation for why the traditional approaches to visualization of model differences do not satisfy all of these concepts.

For example, in text-based visualization, it is hard to provide overview, because the text usually does not fit the display. Modern text-based visualization techniques, for example the technique described in [107] provide means for overview, zooming, and filtering by having a slider on which the position of changed elements in relation to initial models is marked. However, the overview, zooming and filtering in this case are syntax-based and do not provide much insight into the meaning of differences. Furthermore, in order to get insight into the details of selected differences a user needs to spend quite some time interpreting the textual representation of the differences.

In tree-based visualization, for example in [9], getting the overview of the dif-

ferences is still not so easy for larger models because the size of the visible part of the tree is limited by the size of the display. However, the combination of the overview and inherent zooming allows for a much better insight in the meaning of the differences. The filtering in tree-based visualization approaches is also easier than in text approaches, since the hierarchical structure of the tree allows for an easier interpretation of the meaning of differences. However, the filtering still does not provide the means to extract just the required combination of parts of the model and parts of the differences model. Also, the details of the differences are still not easy to comprehend, since the user needs to interpret the tree representation of the differences.

In the diagrammatic visualization approaches, for example in [102], the differences are represented in a natural visualization environment of the metamodel of the models used to calculate the differences. The combination of overview and zoom allows the differences to be examined at the appropriate level of detail, thus providing clear insight into their meaning. For example, a small overview is combined with a larger zoom view, such that the portion of the system that is currently in the focus of the zoom is also outlined in the overview. However, the filtering of differences based on their meaning is still hard; for example, it is hard to extract all element types having a certain property, because these element types might be in different parts of the system, and thus they might only be visualized in different diagrams.

The mentioned examples uncover another reason why traditional approaches scale poorly with the size of the differences model—the traditional approaches use only one visualization technique. Thus, approaches that aim to provide a visualization of model differences that scales well, have to combine multiple techniques.

5.3 Preliminaries

As already mentioned, the three main concerns in the process of comparing models are the representation of, the calculation of, and the processing (e.g., visualization) of model differences. In this section we first describe our approach to the representation of model differences, which specifies the difference presen-

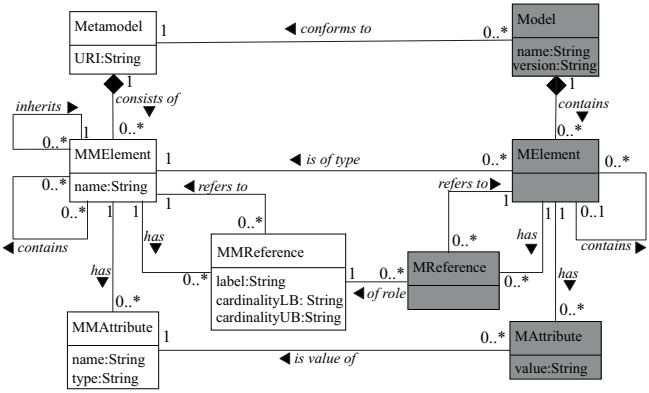

Figure 5.1: Metametamodel that models used in the calculation of differences conform to

tation format used in our visualization technique. This approach is generic and metamodel-independent, and is similar to approaches like EMFCompare [9] or the approach presented in [50]. Next, we briefly describe our approach to the calculation of model differences that produces difference models that conform to the specified presentation format. The details of both approaches can be found in [111].

5.3.1 Representation of model differences

Our approach to the representation of model differences allows those differences to be used in Model Driven Engineering environments. Thus, the differences between two models are represented by a difference model which conforms to a difference metamodel. The difference metamodel is based on the metametamodel that the models (and metamodels) used in the process of calculating differences conform to. This metametamodel is depicted in Figure 5.1. This metametamodel describes both metamodels and models. Metamodels are obtained by instantiating the *Metamodel* element, and models are obtained by instantiating the *Model* element. This metametamodel can be considered as a *domain specific* metametamodel which is geared towards representation of model differences, and not towards general modeling like MOF or Ecore. Thus, it is more comparable to the

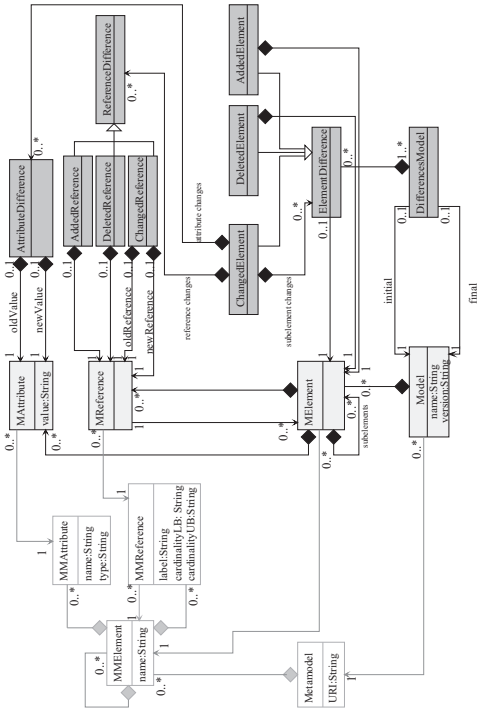

Figure 5.2: Differences metamodel

cores of MOF and Ecore, and does not contain some of the advanced modeling concepts like packages or inheritance. However, the ideas presented using this metametamodel also apply to other, more complete, metametamodels.

The difference metamodel is an extension of the introduced metametamodel and is depicted in Figure 5.2. The difference models are instances of the *DifferencesModel* element. The main building blocks of the difference models are instances of *ChangedElement*, *DeletedElement*, and *AddedElement*. Assuming that the difference model represents the differences between models *A* and *B*, then the instances of the *AddedElement* are elements that are in model *B* and not in model *A*, the instances of the *DeletedElement* are elements that are in model *A* but not in model *B*, and the instances of the *ChangedElement* are elements that

represent the *same entities* in both models but are not structurally identical.

5.3.2 Calculation of model differences

Traditional approaches to the calculation of model differences are based on tree-matching algorithms. These algorithms *match* the nodes of two trees that represent two models being compared and based on this matching the differences are calculated. Several types of matching are recognized: static-identity, signature-based, similarity based or language-specific. Our algorithm for calculating differences is also based on tree-matching algorithms. Unlike traditional approaches that support only one type of matching, our algorithm is defined in such a way to support all four types of matching. In order to allow such a highly configurable calculation process, we extend the differences metametamodel with additional elements. The extended metametamodel is interpreted as a calculation metamodel and is depicted in Figure 5.3.

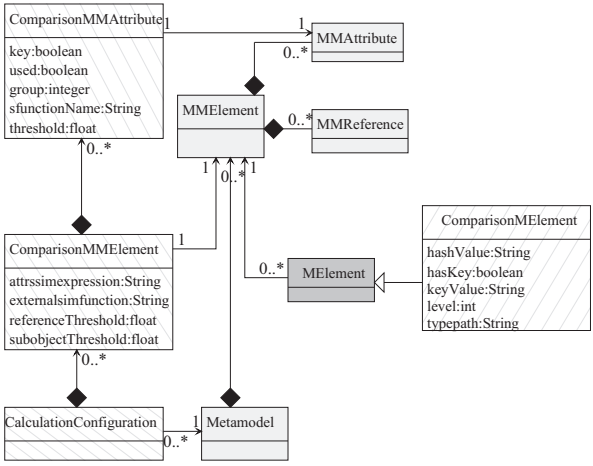

Figure 5.3: Calculation metamodel

Calculation models are used by our algorithm and they have two important features. The first feature is represented by instances of the *CalculationConfiguration* element. This feature of the calculation model offers the possibility of

specifying the metamodel-specific configurations that are used to influence the calculation process of models related to the specific metamodel. Thus, for all models that conform to a specific metamodel, only one calculation configuration needs to be set. The second feature is represented by instances of the *ComparisonMElement*. The instances of the *ComparisonMElement* represent nodes in model trees, and are generated for each model element in a preprocessing step, with the help of the metamodel-specific configuration.

Model comparison algorithm

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

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

In the last step, the calculation of differences is done based on the matchings found. This step consists of three sub-steps. The first sub-step is the calculation of differences in terms of deleted or changed sub-elements and attributes (for all matched and not-matched elements in model A). The second sub-step consists of the calculation of added elements (for all non-matched elements in model B). The third sub-step consists of the calculation of changes in references.

5.4 Differences Visualization

As already noted, our approach to the visualization of model differences extends and combines two other approaches. The first approach involves *polymet-*

ric views, which were first described in [87]. A polymetric view is a lightweight visualization component, which gives insight into a certain aspect of the system by combining simple visualization and metric information. The idea of [87] was used for visualizing model differences in [120]. In the approach of [120], two trees representing two models being compared are represented in a *unified* form. This unified form represents *matched* elements in models being compared as the same node in the tree. This representation allows the definition of metrics related to model differences based on the unified tree. These metrics can then be used to specify views that provide insight in the relation of model differences and model elements.

We follow the approach of [120], and use a unified tree as a representation of both models used in comparison. However, since we do not impose a restriction that a calculation process is included in the visualization process, we do not use calculated similarities between objects as a basis of a metric calculation. The metrics in our case are calculated only by using the initial model (used in calculating the differences) and the differences model. Also, we do not presuppose the significance of *changes*, but this significance can be encoded in the metrics calculation functions.

Our approach to polymetric views is generic, in a sense that we allow user-defined views and metric functions. However, as noted by Lanza et al. [87], since the users rarely define their own views and metrics, we defined a small set of metrics, and by using those metrics we defined a default set of views (based on the set of *cluster* views defined in [87]). The set of metrics we defined is presented in Table 5.1. The metrics apply to instances of MElement, which are also called *objects* in this context. The *subobjects* are instances of MElement contained in another instance of MElement.

Based on this set of metrics, we defined a set of views. For example, the *SYSTEM HOTSPOTS* view specified below defines an overview of the system: all elements in the model are visualized, the width of each element is relative to the number of attributes, the height of each element is relative to the number of references in the element. The color of each element is based on the relative number of changes to the element (the color gradient from white to red is used to represent the relative number of changes), the elements are sorted by the color,

Name	Description
MA	Number of attributes
MR	Number of references
MSO	Number of subobjects
NA	Number of changed attributes
NR	Number of changed references
NSOD	Number of changed direct subobjects
NSOT	Number of changed subobjects which takes into account the transitive closure of the subobject relation
NC	Sum of attribute, reference and subobject changes
RNA	Relative number of changed attributes
RNR	Relative number of changed references
RNSOD	Relative number of changed direct subobjects
RNSOT	Relative number of changed subobjects which takes into account the transitive closure of the subobject relation
RNC	Relative number of sum of attribute, reference and subobject changes
MMName	Encoding of the MMElement instance names

Table 5.1: The defined set of metrics

and the elements are presented in the form of a checker table (a checker table is a simple matrix representation, where elements are visualized in the cells of a matrix based on the specified sort criteria).

- SYSTEM HOTSPOTS - Layout: Checker. Target: Objects. Scope: All. Width: MA. Height: MR. Color: NC. Outline: -. Sort: Color.

The attributes of the views are almost the same as defined in [87]: *Width*, *Height* and *Color* of an icon representing a selected model element. The exception is an extra attribute: *Outline* adapted from [120], which is defined as a colored outline of the icon representing a selected model element. Each of the attributes is related to a defined metric, and this connection is used to provide visual characteristics to the icon representing a model element in a view.

Polymetric views provide a good overview of the changes, the zoom capabilities are also supported by selecting the type of elements to be visualized, and the filtering is done by variations in dimensions and colors of visualized elements. However, the meaning of details of differences is not easily grasped. An example that reveals this problem is depicted in Figure 5.4. In this example, a simplified *INHERITANCE CLASSIFICATION* view of an example model, and a semantically rich view of the same model are presented.

In order to provide better insight to the meaning of the differences details, we extend the polymetric views approach with the possibility of defining special ”views” that expose the details of the differences in a natural (metamodel-specific) way. Thus, for example, a user can select an icon in a polymetric view, and choose to visualize its details in the metamodel-specific way. In software language terminology, the metamodel specific way of visualizing the model corresponds to a *concrete syntax* of a model. The example is given in Figure 5.5.

The second approach borrows the ideas of an ”Open Visualization Framework for Metamodel-Based Modeling Languages” specified in [55], and is also quite similar to the automated approach to generation of model editing environments called EuGENia [14], which allows the users to declaratively map metamodel elements to GMF [18] elements which are used to visualize and edit Eclipse-

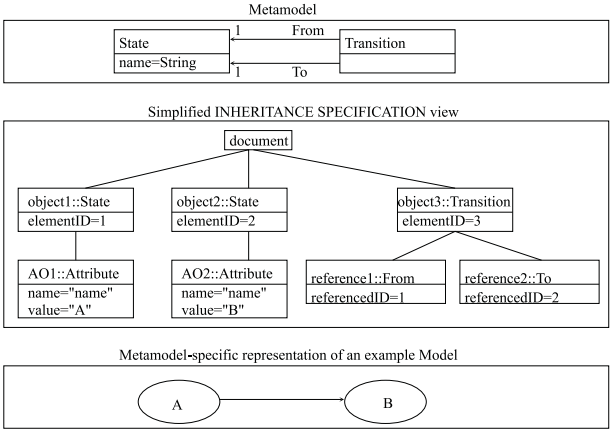

Figure 5.4: An INHERITANCE SPECIFICATION view and the metamodel-specific representation of the same example model

based models [10]. This permits semi-automatic creation of model editors. However, unlike the authors of EuGENia (and GMF) we are not focused on creating complete editors. Our goal is a framework-independent approach to visualizing model differences that fits into the idea of polymetric views.

In our approach we specify a small set of rule types for defining rules to map an arbitrary metamodel onto a graphical metamodel based on *dot* [8]. This allows visualization of models conforming to the mapped metamodel. By using a *unified* representation of models and differences, it also allows the visualization of model differences. Since the *dot* framework is able to automatically layout graphs, we get the layout of the visualized model differences for free. The rules also allow for the definition of a metamodel-specific differences layout. In the following sections we will first present the mapping, the rationale behind it, and the mapping language. Then we will present our approach to the visualization of differences by using the specified mapping.

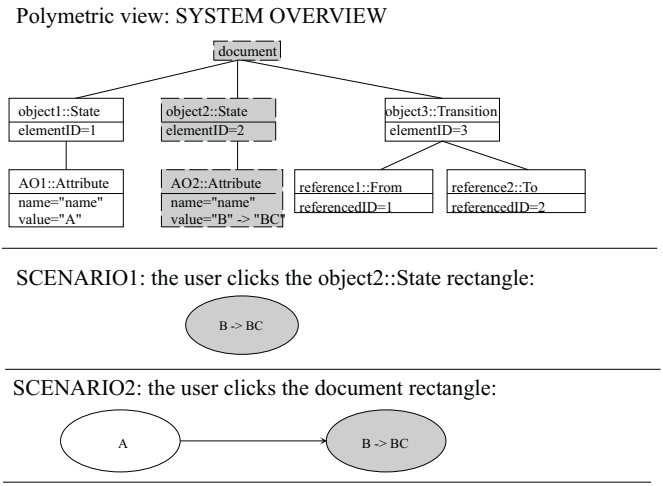

Figure 5.5: Example of combination of polymetric views and metamodel-specific visualization approaches

5.4.1 Metamodel to dot mapping

In order to define a generic metamodel-to-dot mapping, we specify a requirement that all elements and references in a metamodel are identifiable. The example metametamodel that we use in this chapter supports this requirement (see Figure 5.1).

The mapping is represented by a set of rules. Each rule specifies a mapping of one metamodel element to one or more *dot* graphical elements. Each *dot* graphical element is described by a simplified metamodel depicted in Figure 5.6.

We define five rule types. The first rule type defines a mapping of an instance of an *MMElement* into a dot node. All instances of the *MElement* related to the mapped instance of an *MMElement* will be graphically represented by a dot node of shape and size as specified by the mapping. The second rule type defines a mapping of an instance of an *MMElement* into a dot cluster. The dot cluster is a rectangular area that groups a set of dot nodes. This rule enables the creation of a *compositional* hierarchy of elements. The third rule type defines a map-

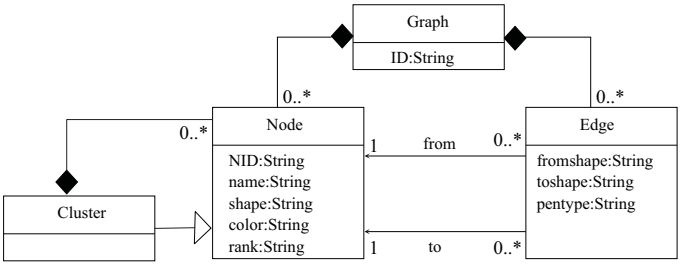

Figure 5.6: Simplified *dot* metamodel

ping of an instance of an *MMElement* into a dot edge. This rule type enables the creation of connections between objects. This can be used to visualize objects that represent, for example, associations or generalizations in UML class diagrams or transitions in UML state machine diagrams. The fourth rule type defines a mapping of an instance of an *MMReference*, into a dot edge. This rule type enables the creation of *tree-like* hierarchical structures. The fifth rule type defines a mapping of an instance of an *MMElement* into a *nexus*. The nexus is a dot node which has one or more incoming dot edges, and one or more outgoing dot edges. The nexus can be used to visualize elements like, for example, pseudostates in UML state machine diagrams. The reasons for specifying these rule types, the details of those rule types, and example mappings, can be found in the Appendix B.

5.4.2 Using the defined mapping to visualize the differences

Consider the differences metamodel as defined in Section 5.3. There are three basic types of differences: added, deleted or changed objects or parts of objects. We will use *coloring* as a way of expressing differences, and thus we assign three colors to these differences. The added objects or parts of objects will be colored green, the deleted objects or parts of objects will be colored red, and the changed objects or parts of objects will be colored blue. The visualization uses both the initial model and the differences model to create a visualization of differences.

The visualization of differences depends both on the mapping, and on the type of difference. We will now describe all the possible combinations of types of differences and mappings of model objects.

If the difference is the addition of an object, and the object is mapped to a node or a cluster, the node or a cluster is presented colored green. If the difference is the addition of an object, and the object is mapped to an edge, the edge is colored green. If the difference is the addition of an object, and the object is mapped to a nexus, the nexus is colored green with all incoming and outgoing edges also colored green.

The coloring principle for the added objects also holds for deleted objects, which are colored red, and for changed objects, which are colored blue.

5.5 Tool

The implemented visualization prototype tool is part of the Java-based framework that also supports the graphical representation of models and metamodels, as well as the graphical representation and the calculation of model differences [30]. In a default setting, the differences are visualized by using a set of predefined polymetric views. This set of predefined views can be changed and extended easily by the users of the tool. The predefined views use a set of predefined metrics. This set of metrics can be extended by new metrics conforming to the predefined interface (new metrics are implemented as Java methods). In order to visualize differences between models conforming to a specific metamodel in a natural way, i.e., in order to use a framework for visualization of metamodel-based languages, a mapping between that metamodel and a *dot* metamodel should be defined by the user. However, this mapping needs to be defined only once for a specific metamodel, and can then be reused for visualizing all model differences obtained by comparing models conforming to that metamodel.

The implemented tool is capable of visualizing the differences between models, completely metamodel independent, as long as models and metamodels provided conform to the metametamodel specified in Section 5.3.1.

To provide more insight into the possibilities of our approach, this section provides a small use-case scenario.

Assume that the first designer has created model A, and that afterwards a second designer has changed that model to a model A'. Next, the first designer would like to inspect the changes to the initial model. In order to do that, he would like to have an overview of the system, such that the elements of the system and the degree of change to each element are given in a form of a checker table, such that he can discern the most changed parts of the system, and he can get an overall impression of changes to the system. Also, he would like to select a changed element and to check the changes to that element in a natural environment for that element (for example to examine the class diagram of the most changed class). Assuming that the models A and A', as well as their metamodel (M_A) conform to the required metametamodel (see Figure 5.1), the first designer should first invoke the differences calculation tool to calculate the differences. The difference calculation tool initially uses a predefined metamodel-independent calculation configuration, however this configuration can be changed to obtain more precise results.

Next, in order to visualize differences in a metamodel-specific way, the first designer needs to define the mapping between the metamodel M_A and the *dot* metamodel. Then, the first designer can invoke the visualization tool and choose an appropriate view (e.g., *SYSTEM HOTSPOTS* view), to get an overview of the differences. Zooming is done by selecting the appropriate type of elements to visualize in a view, and filtering is done by consulting the color of the elements (e.g., more red are more changed elements). Thereafter, the first designer can select a glyph present in this view and visualize it in a metamodel-specific way, in order to obtain details-on-demand of a specific model element.

Two example models A and A', conforming to the metamodel in Figure 5.4 are presented in Figure 5.7.

The visualization of the differences between models A and A' is presented through a series of screenshots of our visualization tool. The first screenshot, shown in Figure 5.8, depicts the initial, tree-like, view on the initial model, with the differences model superimposed on the initial model using a coloring technique.

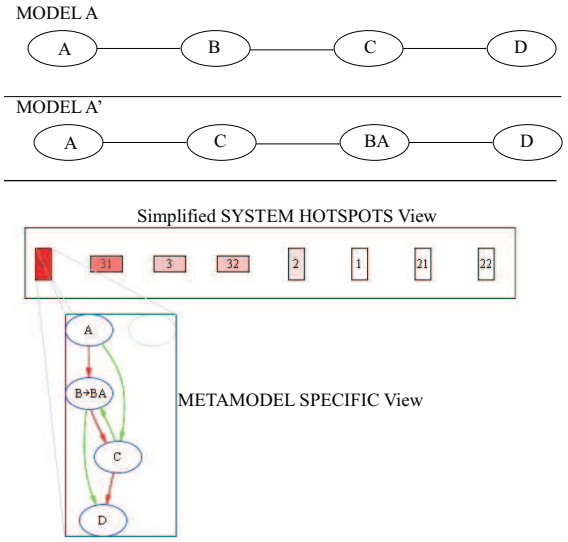

Figure 5.7: Example differences visualization

The second screenshot, shown in Figure 5.9, depicts the *GLOBAL TREE* view. This view depicts the model, tree-like, but now each model element that has been changed has been colored in different hue of red.

The third screenshot, shown in Figure 5.10, depicts the *GLOBAL CHECKER* view. In this view, all elements in the model are presented in the checker table, and all changed elements have been colored in different hue of red. Moreover, the elements have been sorted, in ascending order of changes.

The last screenshot, shown in Figure 5.11, depicts the metamodel dependent view on the changes. In this view, the differences were superimposed on the initial model using a coloring technique.

5.6 Conclusions

5.6.1 Discussion

The traditional difference visualization approaches using only text-based, tree-based or diagrammatic techniques, scale poorly with the size of the difference models. Using polymetric views to visualize model differences, as presented in [120], was a step beyond traditional approaches. However, insight into the meaning of the details of the differences is not easily obtained by using only polymetric views. Thus, we combined polymetric views with a framework for visualizing metamodel-based languages, to obtain more insight into the details of differences. As already noted, a similar approach as ours is described in [102], where a tree-based visualization technique provided by *EMFCompare* is combined with a visualization framework for visualizing model differences by reusing *GMF*. However, since polymetric views efficiently deal with overview, zooming, and filtering, we believe that polymetric views are better suited for providing an overview of larger difference models than the tree-based visualization technique. Also, our approach and accompanying tool are metamodel and framework independent, and thus easily adaptable for a wide spectrum of modeling environments.

5.6.2 Future Work

We are currently performing a validation of our approach in an industrial setting. We intend to use the results of that validation for researching new visualization techniques that can be used specifically for the visualization of model differences. Another possible research direction is investigating the applicability of our approach in a setting of an existing model configuration management system. However, our current approach is focused on differences between two models, but configuration management systems also require a difference function between three models. Thus, research into the applicability of our approach for presenting, calculating, and visualizing differences between three models should be performed first.

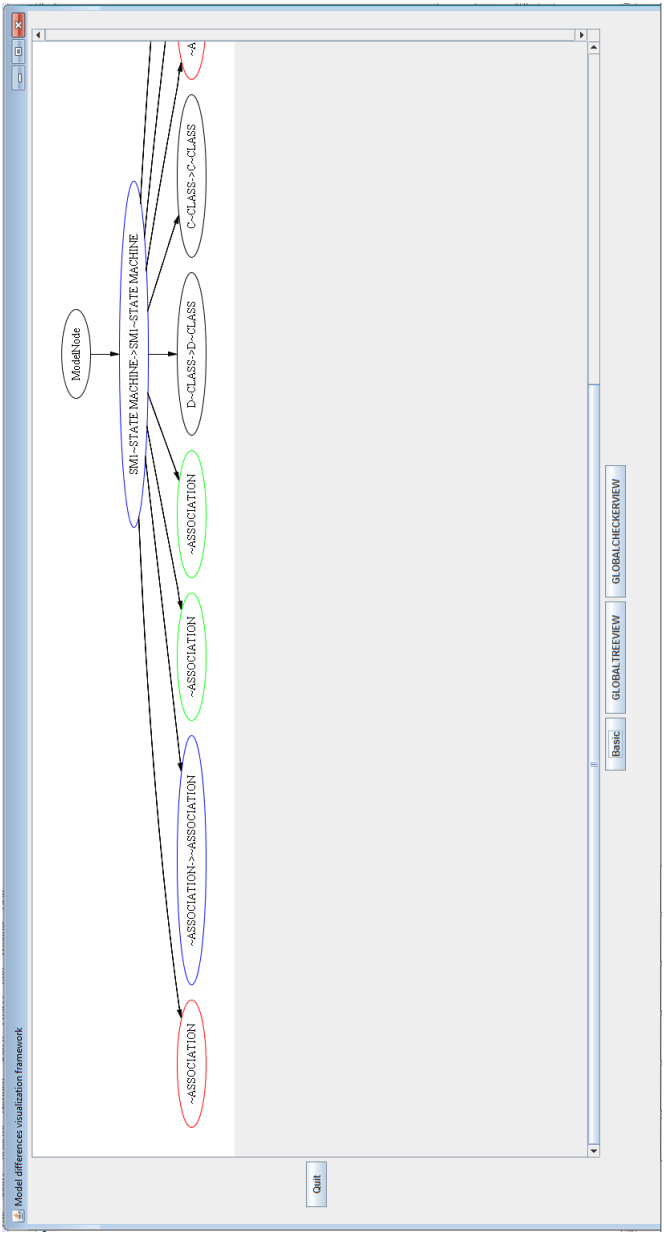

Figure 5.8: Initial view on the initial model, with superimposed differences

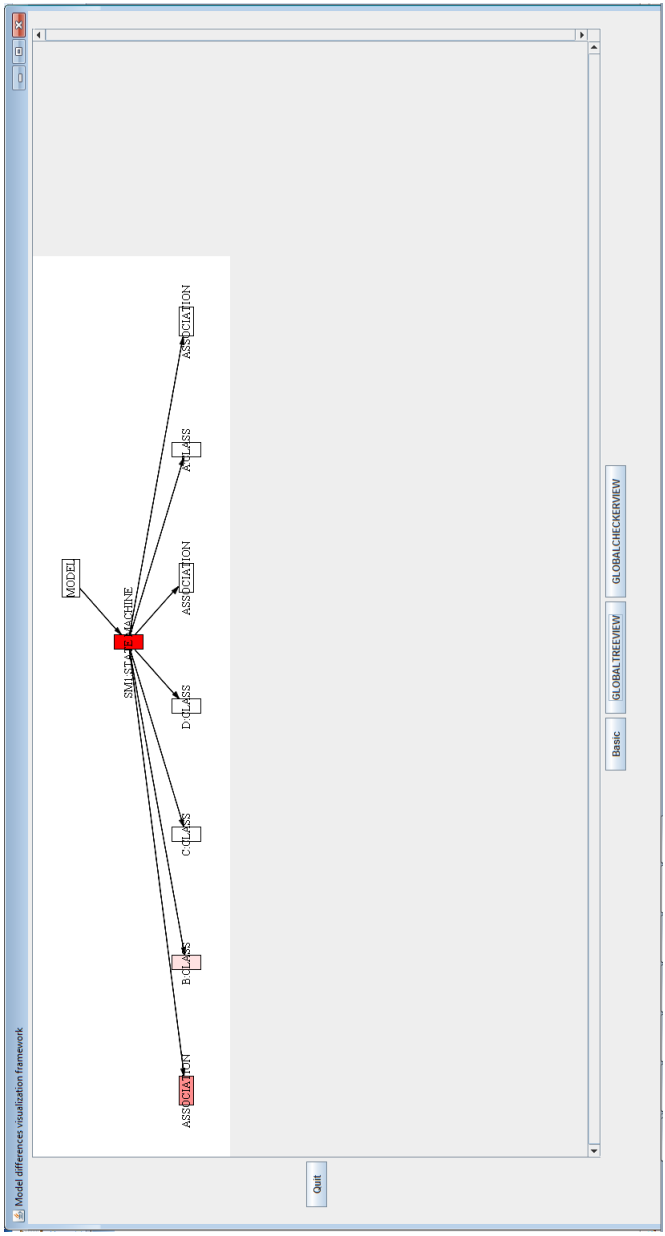

Figure 5.9: GLOBAL TREE view

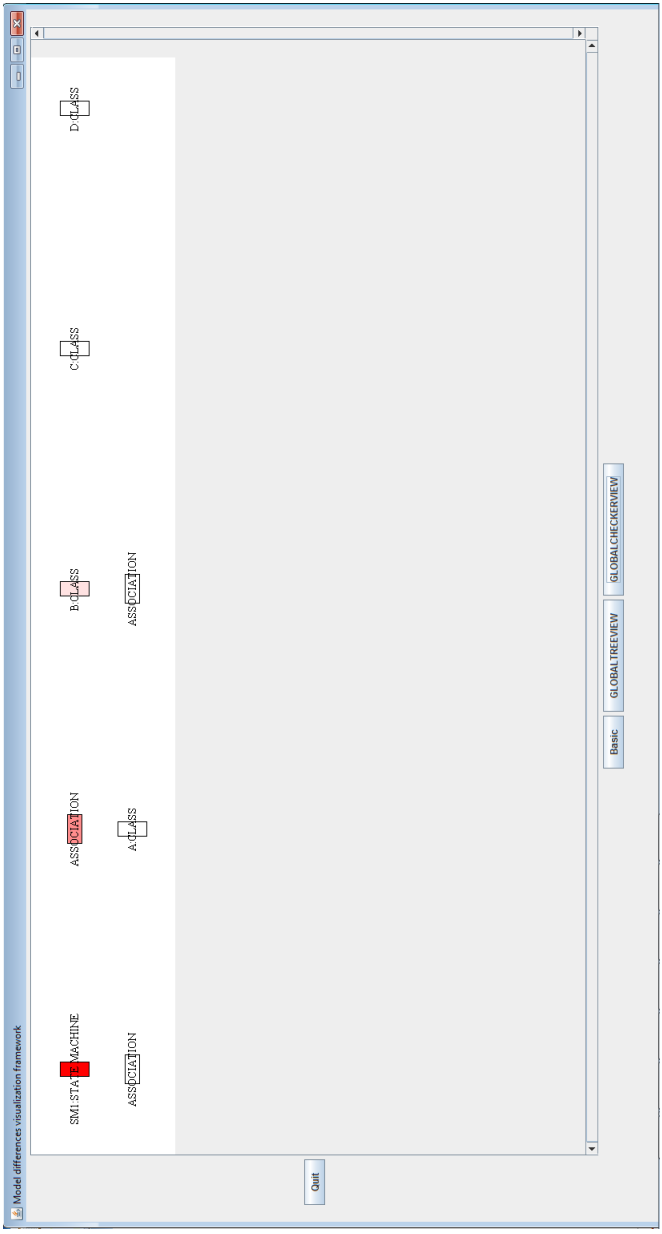

Figure 5.10: GLOBAL CHECKER view

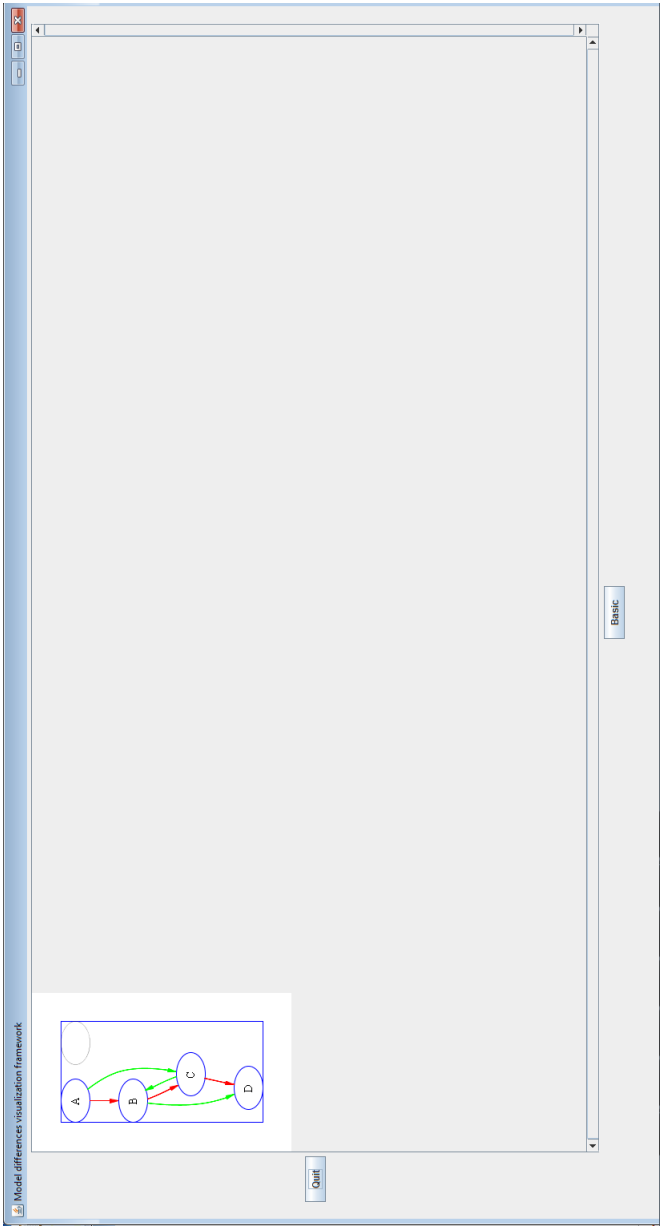

Figure 5.11: Metamodel view on the initial model, with superimposed differences

Chapter 6

A Generic Solution for Syntax-driven Model Co-evolution

This chapter describes our approach to adaptation of models in case their metamodels evolve (also referred to as model co-evolution). In our approach, metamodels are transformed into models conforming to the special metamodel for metamodels (MMfMM) that we developed. This allowed us to use techniques described in Chapter 3 to compare the metamodels, now represented as models. Furthermore, because the syntax of MMfMM is known, all the possible atomic differences between metamodels are also known. Based on this set of possible atomic differences, we have defined a set of rules that can be used with the differences between an initial and an evolved metamodel to co-evolve models conforming to the initial metamodel. In order to validate our approach we have developed a tool for model co-evolution, that is also described in this chapter. Furthermore, we have performed, and described, a series of experiments to test the devel-

oped tool. The results of the experiments show that our approach to model co-evolution behaves as expected.

6.1 Introduction

Model evolution is a frequent research topic in the context of model-driven engineering. Modelers often need to determine the extent and the nature of changes between different versions of the same model. To understand the evolution of a model, modelers compare two versions of that model, and visualize the resulting differences.

Traditionally, models are described as instances of metamodels that, in turn, are instances of a selected metametamodel. Without exception, metametamodels (e.g. MOF [20] or Ecore [11]) allow for the representation of models as hierarchical labeled attributed graphs, i.e. each model can be represented as a tree[1]. The model differences are also considered as trees in [9, 50, 70, 85, 111], and the comparison of models [79, 111] is based on tree comparison techniques [48].

There are two conceptually different types of approaches to the representation and calculation of model differences. In the *state-based* approaches, the model differences are calculated between two states of a model, i.e. between two versions of a model. In the *operation-based* (also called *change-based*) approaches, the model differences are represented by a set of *operations* which when applied to the initial model produce the final model. Thus, in the *operation-based* approaches, all the tools used to develop models must supply the *operations* in a predefined form, while in the *state-based* approaches this is not necessary. Visualization of the model differences is usually accomplished by superimposing the model differences on the old version of a model, and by using different colors to denote different types of differences (e.g. green for added, red for deleted, and blue for changed model elements) [112, 120].

Often, the metamodels also evolve in the modeling process, either during de-

[1]Model elements are nodes of the tree, and edges of the tree are aggregation relations between model elements.

velopment or during maintenance[2]. This raises the question of *co-evolution of models*[3]: how to adapt models conforming to the original version of a metamodel such that they conform to the target (evolved) version of that metamodel? Since metamodels in model-based engineering correspond to languages in language-based engineering, model co-evolution can be compared to the situation in language-based engineering, where a new version of a programming language requires adaptation of the source code written in the old version of a language. Similar problems also exist in database schema evolution, where evolution of a database schema (which corresponds to a metamodel of the underlying data) induces evolution of the related database content.

The basic idea of existing approaches to model co-evolution, which we also adopt here, is: first calculate the differences between an evolved metamodel and an original version of the same metamodel, and then, based on those differences, (semi-)automatically generate model differences. The schematic of our approach is depicted in Figure 6.1.

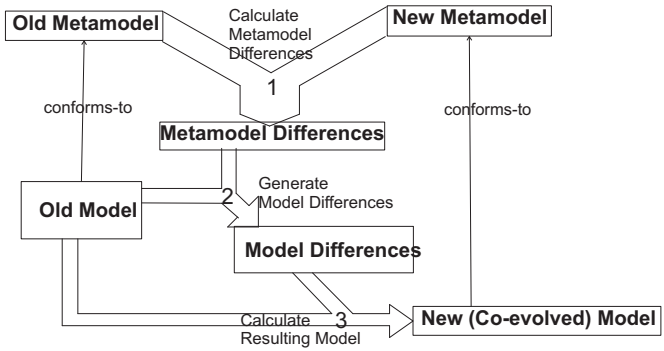

Figure 6.1: The schematic of our approach to co-evolution of models

In this paper we consider model co-evolution in the context of model configuration management systems. Therefore, we have specified a set of requirements that a co-evolution process should satisfy in order to be efficiently usable in such

[2]Similarly to *model difference*, a *metamodel difference* denotes the change set between an old and a new version of a metamodel.

[3]Also called *coupled evolution of models* or *coupled evolution of metamodels and models*.

systems, and we have defined our co-evolution process to satisfy these requirements:

1. The co-evolved models are syntactically correct, i.e., conforming to the new metamodel.
2. The difference between the old model and the new (co-evolved) model is minimal, i.e., only 'necessary' changes are carried through.
3. The co-evolution process allows for (user-defined) extensions to preserve semantic correctness.
4. The co-evolution process itself maximizes automation, i.e., minimizes human intervention, and where intervention is unavoidable it should be well-defined.

In order to satisfy the *first* requirement our process of co-evolution is guided by syntax. We decided to focus on syntax because in co-evolution approaches which deal with *state-based* (meta)model differences, it is very hard to correctly infer the intention of a developer in case of complex changes. Thus, in the *state-based* co-evolution approaches it is much harder to reason about the influence of metamodel changes on models, than in the co-evolution approaches that deal with *operation-based* model differences, where the intent of the model developer is discernible from the nature and the order of the operations supplied by tools. Therefore, in our approach we consider only the syntactic structure and the static semantics of models as a basis of the *automated* part of the co-evolution process, and do not take into account dynamic semantics of models. By dynamic semantics we mean a formal system of rules (e.g. Structured Operational Semantics), that allows reasoning about the behavior of systems represented by models. Thus, in contrast to approaches to database schema evolution, which are geared towards automatic resolution of semantic issues (i.e. retaining the relations between data items), but are constrained only to schema evolution, we loosen the requirement of *automatic resolution* of semantic issues, in order to be more generic and to support arbitrary metamodels. Hence, our approach can support the co-evolution of databases, ontologies, state machines, petri-nets, etc. Nonetheless, as specified in the *third* requirement, it is possible to define user extensions to ensure the semantic correctness of the co-evolved models. This means that advanced algorithms for schema evolution or petri-net evolution can

be applied as extensions to our approach.

The *second* requirement states that the co-evolved model should be changed as little as possible to conform to the new metamodel, thus allowing efficient implementation of our approach in configuration management systems.

The *fourth* requirement states that the co-evolution process should be as automatic as possible, and that the reasons for, and extent of, human interventions should be well-specified and minimized. In this regard, existing approaches to model co-evolution [49, 52, 62, 67, 71, 118] classify metamodel differences based on how they affect both the co-evolving models and the possibility to automate the co-evolution process. Three groups can be distinguished: non-breaking differences, breaking but resolvable differences, and breaking but non-resolvable differences. *Non-breaking differences* (NBD) to a metamodel do not require any change in the models. *Breaking but resolvable differences* (BRD) require a transformation of the model, which can be automated. *Breaking but non-resolvable differences* (BNRD) require user intervention and are almost impossible to automate. Next, the existing approaches define, depending on which metametamodel is used (e.g. MOF or Ecore), all possible metamodel differences, and relate these differences to the three defined groups. Furthermore, the non-breaking differences, and breaking but resolvable differences, are used to automatically generate model differences, and the breaking but non-resolvable differences are resolved with the help of a human.

We split the possible differences into four groups based on their influence on the *syntactic structure* of co-evolving models and based on the possible automation of the co-evolution process. In particular the group of *breaking but non-resolvable* differences is split into two groups: *breaking and semi-resolvable differences* (BSRD) and *breaking and human-resolvable differences* (BHRD). *Breaking and semi-resolvable differences* are differences which can be automatically resolved by configuring the co-evolution process. These differences also encompass the semantic differences which can be resolved by taking into account the semantics of the models. *Breaking and human-resolvable differences* can only be resolved by a user in a differences-resolution environment and cannot be fully automated. For example, if a reference, which has a lower bound of 1, is added to a metamodel, in order to obtain the correct resulting models,

concerning the intention of a metamodel developer, a user needs to connect the correct objects in models.

As already mentioned, although our approach is not geared towards automatic resolution of semantic problems, the specified tool architecture is extensible and can be extended to deal with the semantic issues. For example, a logic-based conflict resolver such as Aleph used in [51], a generic model transformation method like Viatra [33], or, in case of database schemas, a database schema matching algorithm like Cupid [91] can be used to resolve possible semantic problems.

The outline of the rest of the chapter is as follows. In Section 6.2, we discuss some preliminaries necessary to understand our approach. Then, in Section 6.3, we discuss the evolution of metamodels. Next, in Section 6.4 we discuss the process of co-evolution of models. Furthermore, we describe the tool we built that faithfully implements our approach, and we describe an experiment we performed to validate our approach. Finally, in Section 6.6, we conclude the paper and give some directions for further research.

6.2 Preliminaries

In this section we give some preliminaries necessary for understanding our method of model co-evolution. We first describe a special *domain-specific* metametamodel which we use in describing our method. This metametamodel is simple, but allows formal reasoning on metamodels, models, and their relation. Next, we describe a generic differences metamodel, which is based on the described metametamodel. This differences metamodel is used to capture the differences between two models, and, in our approach, also the differences between two metamodels[4].

[4]The details of both metametamodel and the differences model can be found in Chapter 3.

6.2.1 Domain-Specific Metametamodel

We approached the problem of generic model differences by designing a domain-specific metametamodel, that exposes not only the details of metamodels, but also the details of models, and the relations between metamodels and models. Metamodels are obtained by instantiating the *Metamodel* element (non colored elements in Figure 6.2), and models are obtained by instantiating the *Model* element (grey elements in Figure 6.2). Each metamodel can contain a set of named elements. Each of these elements can contain named and typed attributes, and labeled references to other metamodel elements. Each model can contain a set of model elements, that must conform to metamodel elements. Moreover, each model element can contain attribute instances (containing values), and reference instances (referencing other model elements). Unlike in traditional metametamodeling approaches (e.g. MOF or Ecore), in our approach models are not considered instances of metamodels, but models only *conform-to* metamodels. However, both models, metamodels, and their relationships, are instances of the introduced metametamodel. Notice that although our metametamodel is designed for a specific domain of model differences, it allows for description of labeled attributed graphs, and thus is quite generic (i.e. it allows for description of all graph-based systems). For example, we have developed transformations from metamodels conforming to Ecore, and models conforming to those metamodels, to our formalism. This makes it possible to use our co-evolution approach with the Ecore-based metamodels and models. The architecture of our metametamodel allows the specification of a metamodel-independent differences metamodel [111], which is discussed in the following section.

6.2.2 Model differences

Our approach to the representation of model differences satisfies all of the requirements specified in [50], These requirements allow model differences to be seamlessly used in model configuration management systems. The differences between two models are represented by a differences model that conforms to a differences metamodel. The differences metamodel is an extension of the metametamodel introduced in the previous section and is depicted in Figure 6.3. Differences models are instances of the *DifferencesModel* element. The build-

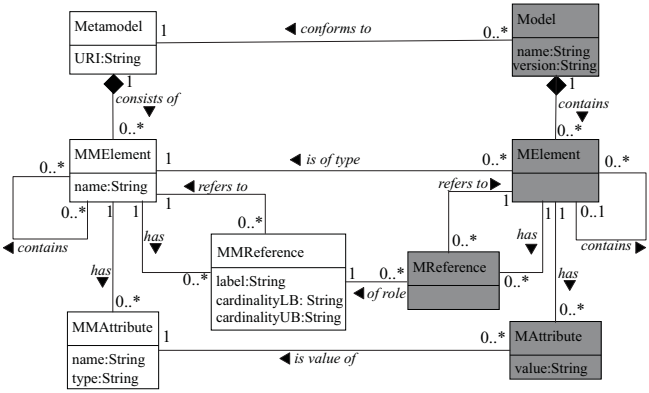

Figure 6.2: Metametamodel

ing blocks of the differences models are instances of *ChangedElement*, *DeletedElement*, *AddedElement*, and *MovedElement*. Assuming that the differences model represents the differences between models *A* and *B*, then the instances of the *AddedElement* are elements that are in model *B* and not in model *A*, the instances of the *DeletedElement* are elements that are in model *A* but not in model *B*, and the instances of the *ChangedElement* are elements that represent the *same entities* in both models but are not structurally identical. Since a differences model contains only references to models, this differences metamodel is generic (metamodel-independent).

6.3 Metamodel Evolution

Traditional approaches to metamodel evolution define special mechanisms for representing, calculating and visualizing metamodel differences. These methods are usually based on techniques for representing, calculating and visualizing model differences, but there is a clear separation between metamodels and models, and thus also between metamodel differences and model differences.

In our approach, the techniques for representing, calculating and visualizing model differences are applied directly to metamodel differences. Our key idea is

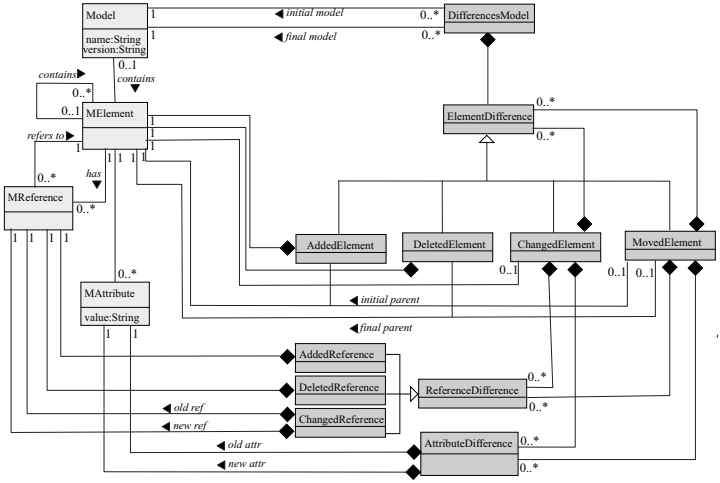

Figure 6.3: Differences metamodel

to represent metamodels as models conforming to a special metamodel. In this way, all the techniques for model comparison can be directly applied to metamodel comparison.

In order to represent metamodels as models, we define a special metamodel for metamodels (MMfMM). The metamodels can now be interpreted as (i.e. transformed to) the models conforming to the MMfMM. Consequently, the differences between metamodels are obtained by transforming metamodels to models, and by calculating the differences between the resulting models. This approach is particulary useful in the context of a model configuration management systems, because it allows a unified treatment of models and metamodels.

In the next section we describe our metamodel for metamodels (MMfMM). By consulting this metamodel, it is possible to specify all the possible types of metamodel differences, and their influence on co-evolving models, which is discussed in Section 6.3.2.

6.3.1 Metamodel for metamodels - MMfMM

In this section we discuss a metamodel for metamodels (MMfMM), depicted in Figure 6.4. Since MMfMM is a metamodel, it is an instance of the *Metamodel* el-

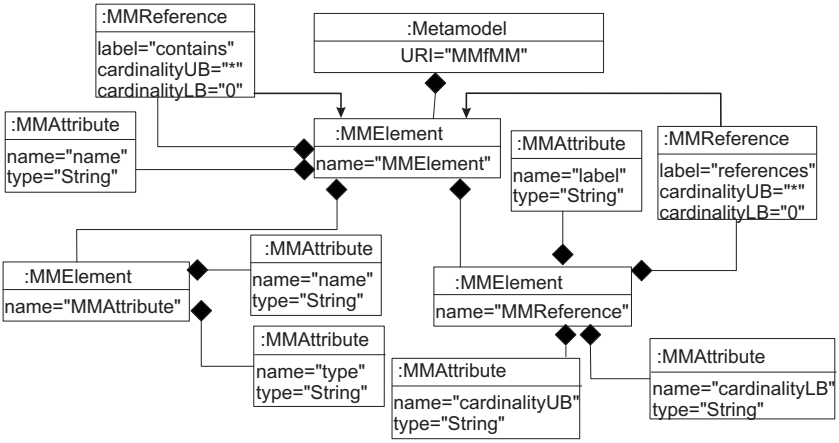

Figure 6.4: A metamodel for metamodels - MMfMM

ement from our *domain-specific* metametamodel (depicted in Figure 6.2). Models that conform to the MMfMM represent metamodels. Thus, each metamodel has two representations: its *natural* representation (instance of the *Metamodel* element), and a *transformed* representation (instance of the *Model* element that conforms to the MMfMM) [5]. However, we designed MMfMM in such a way that a transformation from a *natural* representation of a metamodel to a *transformed* representation is trivial. For example, an MMfMM element named *MMElement* represents metamodel elements. Elements in models that are instances of MMfMM, and that conform to MMfMM element named *MMElement* represent metamodel elements.

A *natural* representation of an example metamodel, and a *transformed* representation of the same metamodel, are depicted in Figure 6.5.

[5] In EMF terminology, MMfMM corresponds to a metamodel *Ecore.ecore*. *Ecore.ecore* is an Ecore-based metamodel that allows for the creation of Ecore-based models such that there is a bijection between any of those models and an Ecore-based metamodel.

A natural representation of a metamodel (top left part of the Figure 6.5) is named *example*, and has two metamodel elements named *State* and *Transition*. Both the State element, and the Transition element, have an attribute Name of type String. Moreover, a Transition element has a reference that has a label Connects. In the transformed (i.e. model) representation of a metamodel (lower right part of the Figure 6.5), the rectangles represent instances of model elements conforming to specific MMfMM elements. The labels of those rectangles can be split into two parts, before and after the token ":". The first part denotes the MMfMM element that the element represented by the rectangle conforms to. The second part represents the type of the model element (*MElement, MAttribute or MReference*). For example, a metamodel element named State is represented with one model element, and with one attribute of that model element that has a value *State* (dark grey part of Figure 6.5). The attribute Name of a metamodel element named State is also represented with one model element. However, this element has two attributes having values Name and String, representing the name and type of the Name attribute.

6.3.2 Metamodel Differences

As already mentioned, in this chapter we focus on automatic processing of syntactic changes (differences) to metamodels. The list of all detectable metamodel differences, and the consequences of these differences are given in Appendix C. In some cases we mention the relation of differences to the (static) semantics of models, and these relations guided our reasoning in many cases. However, since we did not choose any semantic formalism for interpreting the behavior of models conforming to a certain metamodel, we did not formally reason about semantics.

6.4 Model Co-evolution

In this section we present a method for calculating model differences, based on metamodel differences.

In order to obtain the model differences from the metamodel differences, a nec-

essary prerequisite is the existence of formal methods for the representation of metamodels, models and model differences, as well as a method for the calculation of model differences. As mentioned before, without loss of generality we use the metametamodel depicted in Figure 6.2 for the representation of metamodels and models, and a metamodel for the representation of model differences depicted in Figure 6.3.

Next, we assume that the differences between the *evolved* and *original* metamodel have been calculated and are presented as a differences model labeled *differences*. Then, for each model M that conforms to the *original* metamodel, the algorithm described in the following section can be used to calculate the differences model DM, that can be used to patch model M to obtain a new (evolved) model M' that conforms to the *evolved* metamodel.

6.4.1 Model Differences Calculation Algorithm

The calculation algorithm is an implementation of the causal relations between metamodel differences and model differences described in Appendix C. The algorithm traverses the graph representation of a model (actually a tree representation, where the edges are instances of the containment relation, is traversed), and for each model element checks whether the metamodel element that that element conforms to has changed. If this was the case, then, based on the changes to the metamodel element, the model differences for that particular element are generated, otherwise nothing happens.

For solving *breaking and human-resolvable differences* we introduced two special functions in the algorithm. The first of these functions is labeled *warningrequest(name, id)*. This function first checks for the presence of conflicts (*breaking and human-resolvable differences*) of the specified *name* in all model elements that conform to the metamodel element of the specified *id*. If there are no conflicts then the function terminates, and if there are conflicts, an environment for manual conflict resolution is started. This function is used in case of possible conflicts. For example, if the bounds of a reference change, this function checks if the model is in a conflicting state, and, if this is the case, starts an environment for manual conflict resolution. The second function is *conflictrequest(name, id)*.

This function denotes that there is a conflict, having a *name* as specified in the argument of the function, and that it is necessary to start an environment for manual conflict resolution, for all model elements that conform to the metamodel element identified by the specified *id* argument. This function is used in case of affirmed conflicts, for example if the type of a reference changes.

6.4.2 Validation

In order to validate our co-evolution method we built a tool that faithfully implements our method, and we systematically tested this tool with a large set of metamodels and models.

The tool consists of two parts. The first part is responsible for the completely automatic transformation of models by considering non-breaking differences, or the breaking differences which are resolvable by providing a configuration file. The second part is a graphical application, that allows manual resolution in case of breaking changes which are not resolvable automatically. The tool is extensible, and thus users can define additional (e.g. metamodel specific) transformations in order to solve semantic issues that may arise during the co-evolution process. The tool can be configured to call the user-defined transformation functions before, during, or after the part of the co-evolution process that is fully automated.

Our goals in testing the tool were:

- Assessing the capability of a tool in detecting metamodel differences;
- Assessing the functional correctness of a tool in cases of both automatic and semi-automatic processing of differences;
- Assessing the extent of user involvement in adaptation of a larger set of co-evolved models.

For the testing we selected 10 metamodels, and for each metamodel 10 conforming models, giving rise to one hundred models altogether[6]. In order to make

[6]The metamodels used in the tests are generated by using a metamodel-generator tool that we

our experiment transparent, we decided to co-evolve the selected models by using the co-evolution scenarios specified in previous research in this area. For this reason, we selected 9 operations from the set of 61 co-evolution operations defined in [72], and we applied each operation to each metamodel, thus obtaining 90 co-evolved metamodels. These operations were selected such that they ensure coverage of all cases of possible resolution scenarios as specified in Section 6.3.2. Next, we applied our co-evolution tool to each evolved metamodel, co-evolving models accordingly.

For each operation we measured: the number of metamodels for which the tool correctly detected the co-evolution operation, the number of fully automatically co-evolved models, the number of semi-automatically co-evolved models, the number of models that need to be manually co-evolved, and the number of models that did not need to change. The results are given in Table 6.1.

Operation	Automatically adapted models	Semi-automatically adapted models	Manually adapted models	Unaffected models	Total models
Create Class	0	0	0	100	100
Create Attribute	0	80	0	20	100
Create Reference	0	0	100	0	100
Delete Reference	100	0	0	0	100
Rename Attribute	0	0	0	100	100
Make Reference Composite	0	100	0	0	100
Change Attribute Type	0	90	0	10	100
Move Feature Over Reference	0	73	0	27	100
Reference To Class	0	74	0	26	100
TOTALS	100	417	100	283	900

Table 6.1: Model co-evolution results

The interpretation of the results is as follows: *Create Class* and *Rename Attribute* operations are completely automated, and the models do not need adaptation. *Delete Reference* operation is also completely automated, but models are affected. *Create Reference* operation requires user intervention in specifying possible instances of the created reference. *Create Attribute* operation is semi-automated by defining the configuration entry specifying the default value of the created attribute. *Change Attribute Type* operation is semi-automated by defin-

developed, and models used in the tests are generated by using a model-generator tool that we developed. The output metamodels and models are Ecore-based, but are transformed to our metamodeling formalism for the purpose of this validation study.

ing the configuration entry specifying a function for transforming the values of attributes of the original type, to attributes of the new type. *Make Reference Composite*, *Move Feature Over Reference* and *Reference To Class* operations are semi-automated by specifying specific model transformations that deal with semantic issues of these operations. In particular, these operations are built up of many atomic metamodel differences, thus detecting these operations requires pattern matching on model differences[7]. Furthermore, the resolution of these operations requires more complex algorithms than the ones initially provided by a tool.

Our conclusion is that while most models require some form of intervention, this intervention can usually be specified on a per-operation, or on a per-metamodel basis, and not on a per-model basis.

The developed tool, as well as the test metamodels and models, are available online [31][8].

6.5 Related work

Our approach is applicable in case of a *state-based* representation and calculation of model differences [111]. In contrast to approaches that deal with *operation-based* representation and calculation of model differences, such as COPE [70], our approach can also be used for modeling tools that have as output only complete (meta)models and not the set of *operations*. Thus, our approach is most useful if a company uses multiple tools for managing its models. However, if a company uses only one tool for managing all its models, and if that tool can provide operations, an operation-based approach would be preferred.

In our approach, we represent metamodels as models conforming to a metamodel specifically designed for this purpose. Thus, metamodels need to be transformed to equivalent models in order to be used in this manner. The first advantage of our approach with respect to the existing approaches featuring *state-based* model co-evolution is that we do not need to invent a special representation mechanism

[7] In a state-based approach to model differences, which is employed by us.
[8] http://www.win.tue.nl/~zprotic/coevol.html

for metamodel differences, but we represent the metamodel differences as model differences. This allows us to use generic techniques for the representation and calculation of model differences as described in [111], to represent and calculate *meta*model differences. For example in [50, 51], for each metamodel a custom differences metamodel must be specified, whereas in our approach only one differences metamodel is used. Furthermore, our differences metamodel provides a more detailed representation of model differences than, for example, the ones used in [50, 51] (for details see [111]).

Another advantage of our approach is that, since our technique for representing (and calculating) differences is state-based, it does not require special modeling-tool support like operation-based approaches [67, 71], but can be used also with the tools that provide this support.

Furthermore, most existing co-evolution approaches [62, 63], use a single heuristic algorithm for metamodel comparison, where we reuse a generic declarative model-differences calculation algorithm, which is based on tree-comparison techniques, and can be configured such that it does not use heuristics at all [111]. Therefore, in our approach it is possible to easily configure the comparison algorithm, such that it suits the needs of the users.

Finally, we introduce a metametamodel which involves only two metamodeling levels. Because of this we do not require the use of higher-order model transformations for calculating co-evolved model differences [50, 51, 62], but the differences are obtained by an ordinary, first-order model transformation. The advantage of this is that the tool based on our co-evolution approach is easy to build and maintain.

6.6 Conclusions

In this paper we define a method to support the co-evolution of models as induced by the evolution of metamodels. Our main contributions are:

- We show that by representing metamodels as models conforming to a special metamodel, existing techniques for representing and calculating

model differences can be directly applied to calculation of metamodel differences;

- We show that the group of *breaking and non-resolvable metamodel differences* can be further split into two sub-groups based on further possibilities for automation of the resolution process;
- We show that it is possible to have only one, generic, transformation for co-evolving models, which is an improvement to the previous approaches where higher-order transformations were employed;
- We execute a large validation study, showing that it is possible to automate most of the co-evolution process, and that for only a small percentage of changes to metamodels, the co-evolution requires manual intervention.

Our method ensures syntactic correctness of the resulting models. Ensuring semantic correctness of the co-evolved models is supported by providing an extension mechanism for user-defined transformation functions. An example of a semantic issue that can be solved by a user-defined transformation is the introduction of an attribute in a metamodel element whose value in the corresponding model element is to be obtained by combining multiple values of attributes in other model elements.

Since our method uses a *state-based* approach to representation and calculation of model differences, and since it is independent of a specific framework and (meta)metamodel, it is directly applicable in an industrial context for companies that use a variety of tools and that would like to co-evolve models developed with those tools. The stand-alone tool that we developed supports this claim.

Future work includes conducting an even larger and more thorough case study based on an industrial case. Furthermore, it would be interesting to adapt our approach to more popular metamodeling formalisms like MOF [20] or EMF [11].

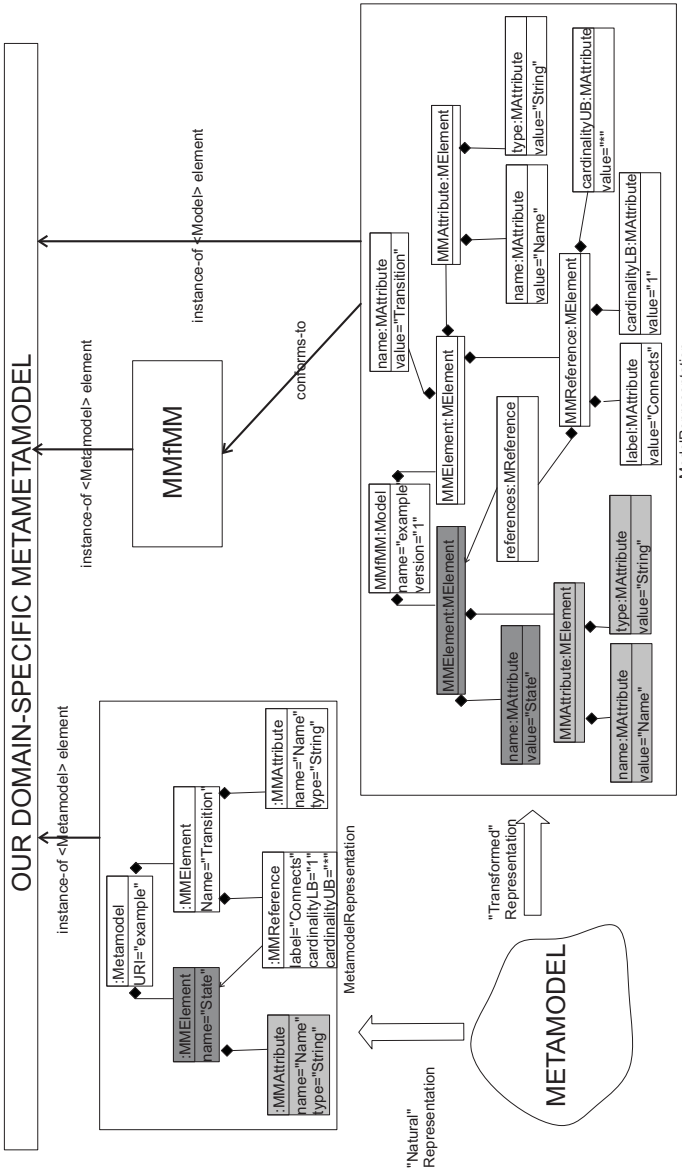

Figure 6.5: Example metamodel in both the natural and the transformed form

Chapter 7

Conclusions

7.1 Contributions

This dissertation addresses three problems in the field of model management: model comparison, model differences visualization, and model co-evolution. In the next three sections we will summarize these problems, and we will discuss our solutions to these problems.

7.1.1 Solution to the model comparison problem

The first problem discussed is the problem of model comparison, that was expressed in Chapter 1 by the following research question:

Research Question 1. *How can the quality of methods and tools for model comparison be improved?*

This research question was addressed in Chapters 2, 3, and 4 of this dissertation.

In Chapter 3 we discussed existing methods for comparing models and for representing the difference between compared models, and we exposed aspects of those methods that can be improved. In particular, the way to improve methods for the representation of model differences, is to specify a metamodel of model differences clearly, rigorously, and unambiguously. However, existing modeling formalisms, such as Ecore or MOF, are inadequately equipped for this. Thus, we first specified a new modeling framework, that is conceptually similar to existing modeling frameworks, but that is equipped with means of expressing the metamodel of model differences as required. This new framework, as well as the metamodel of model differences we developed using this framework, was briefly described in Chapter 3. However, since we consider this new framework as an important contribution, we described the inadequacies of existing modeling frameworks, the new modeling framework, and the metamodel for model differences, in more detail in Chapter 2.

In order to improve methods for calculating model differences, the algorithms for model differences calculation should be made more flexible. For instance, existing algorithms for calculating model differences are all based on (only) one of the four approaches for model matching: static-identity based, signature-based, similarity-based or language specific. However, it might be the case that for one metamodel a signature-based approach is more applicable, and that for another metamodel a similarity-based approach is more applicable. Thus, we defined a configurable, similarity-based, algorithm, that enables all of the four model matching approaches to be used. We achieved this by designing the calculation algorithm and the configuration metamodel in parallel, such that the combination of the configuration models and the algorithm overcome the limitations of similarity-based comparison, and allow all other model matching types to be used. The developed algorithm and the configuration metamodel are described in Chapter 3.

While in Chapter 3 we described conceptual improvements to existing methods for model comparison, in Chapter 4 we provide an experimental validation of introduced improvements. In order to perform this validation we first developed a tool for model comparison based on the algorithm we specified. Next, we de-

fined a method for assessing the quality of model comparison tools, we defined a set of experimental data, and we applied the defined data, as specified by the defined method, to the developed tool. Moreover, we applied the defined experimental data, as specified by the developed method, to another, commercial, tool for model comparison, in order to asses the developed assessment method also. The results show that the comparison tool we developed (and with it our method) is of similar, or of higher quality, than the industrial tool.

7.1.2 Solution to the model differences visualization problem

The second problem addressed in this dissertation is the problem of visualization of model differences, that was expressed in Chapter 1 by the following research question:

Research Question 2. *How to improve existing methods and tools for visualization of model differences?*

This research question was answered in Chapter 5 of this dissertation. In particular, in Chapter 5 we first review existing methods for visualizing model differences. Next, we argued that model differences should be treated as a large-scale information content, and that in order to improve existing methods for visualizing model differences, they should be combined with methods for visualizing large-scale information content. In this respect, we described our approach that combines two existing visualization techniques, in a technique that is better for visualization of model differences than both combined techniques. We validated our approach by building a tool for visualization of model differences.

7.1.3 Solution to the model co-evolution problem

The third problem addressed in this dissertation is the problem of model co-evolution, that was expressed in Chapter 1 by the following research question:

Research Question 3. *How to improve existing methods and tools for adapting models in case their metamodels evolve (i.e. how to co-evolve models)?*

This research question was answered in Chapter 6 of this dissertation. In particular, in Chapter 6 we first reviewed existing methods for co-evolving models and proposed several improvements to these methods. The first improvement we proposed is based on the fact that existing methods leverage ad-hoc comparison algorithms to compare metamodels. We proposed to transform metamodels to models, by a bijective model transformation. This allowed us to use existing methods and techniques for model comparison to compare metamodels (now represented as models). The second improvement is related to the fact that most of the existing methods leverage higher-order model transformations to co-evolve models. However, higher-order model transformations work best only when it is possible to fully automate the co-evolution process. In other cases, e.g., in case that different metamodel elements require different co-evolution strategies, higher order model transformations are not the best option. In order to improve this aspect of existing methods, we specified means of using the ordinary model transformations to co-evolve models. This is possible because, in our approach, metamodels are represented as models, conforming to a metamodel for metamodels (MMfMM). Thus, since all the atomic differences in metamodels (now represented as models conforming to MMfMM) are easily distinguishable, it is possible to define a transformation that takes as an input a model of differences between an initial and an evolved metamodel (represented as models), and that transforms (i.e. evolves) models conforming to the initial metamodel based on the found differences. We validated our approach by building a tool for co-evolution of models, by evolving 10 metamodels (each metamodel was evolved differently), and by co-evolving 50 models conforming to each of those 10 metamodels (500 models in total).

7.2 An overview of the related work

In this section we will give an overview of the work related to model comparison and model co-evolution. Note that we discuss the related work from the perspective of its possible applicability in *model configuration management* and the relation to our work.

7.2.1 Model comparison

Recall that in this dissertation we focus on *2-way model comparison*. In the context of model configuration management, there are two interpretations of a term "2-way model comparison". In the first interpretation, 2-way model comparison is the comparison of two models (*initial* and *target*), to obtain a difference between those models that takes the least memory. This interpretation is aimed at efficient storage of models, which is paramount in configuration management systems. In the second interpretation, model comparison involves discovering and presenting the difference between two models (*initial* and *target*), such that the discovered difference is as close as possible to the difference that a human user applied (or would apply) to the initial model to obtain the target model. In this case, it is assumed that the initial and the target model are evolutionary related (either the initial model is an ancestor of the target model, or that the initial and target model have the same ancestor). In this dissertation we discussed model comparison related to the second interpretation, and in this section we discuss work related to this second interpretation.

Three main aspects of model comparison are:

- Representation of the difference between models
- Calculation of the difference between models
  - Matching models
  - Creation of the difference model
- Visualization of the difference between models

In other software engineering disciplines (other than MDSE) there are no strongly related approaches to representation and visualization of differences between models. Therefore we discussed the work related to these two aspects in the chapters where we discussed our approaches to representation and visualization of model differences (Chapter 3 and Chapter 5, respectively). Moreover, the creation of a difference model, after the models have been matched, is straightforward, as described in Section 3.3.3. Thus in this section we mainly discuss approaches related to *model matching*. The *model matching* problem, discussed

in Chapter 1, can also be formulated as: given two models *initial* and *target*, determine all pairs of model elements, one from *initial* and the other from *target*, that represent the same entity in the modeled system.

We first discuss the cognitive aspects of similarities between entities. Thereafter, we discuss the relation of model matching to the field of ontologies, in particular ontology matching. An ontology represents conceptualizations of a domain (which is a subset of the entire corpus of knowledge), that can be used to reason about that domain [66]. For example, one ontology may consist of a concept named *Person* connected by a relation named <*lives-at*> to a concept named *Address*, while another ontology may consist of a concept named *Individual* connected by a relation named <*lives-at*> to a concept named *HomeAdress*. Ontology matching denotes matching of the same concepts in different ontologies. The problem in ontology matching is that the same concepts may have different representations in different ontologies. For instance, in the previous example, a correct matching consists of a match of the concept named *Person* to the concept named *Individual*, as well as a match of the concept named *Address* to the concept named *HomeAdress*. The field of ontology matching is strongly related to the field of model matching because the methods and techniques used to match concepts present in ontologies are very similar to the methods and techniques used to match (parts of) model elements.

Next, we discuss the relation of model matching to the field of schema matching. A schema is a term from a database community that describes a blueprint of a database. Schemas contain the definitions of the database tables, the definitions of the attributes of those tables, the definitions of relations between tables, and the constraints imposed upon entities in the schema. Schema matching denotes matching the elements of two schemas that describe the same real-world entities. For example, a table named *Person* in one schema may be matched to a table named *Individual* in another schema. The field of schema matching is strongly related to the field of ontology matching, and thus it is strongly related to the field of model matching.

Thereafter, we will discuss the methods and approaches in the field of tree comparison. Trees are directed acyclic graphs, that have one root node. All traditional metametamodels (MOF, Ecore) define the aggregation relation, which

specifies that each model element may have only one parent (i.e. each model element must be contained in only one other element). Thus, models can be transformed to trees by a projection defined using the aggregation relation, and most MDSE approaches to model comparison use the tree structure inherent to models to define the model matching algorithms. Therefore the field of tree comparison is strongly related to the field of model matching.

Finally, we give an overview of the research approaches to model comparison in the MDSE community.

Cognitive aspects and relation to ontologies

A common question in cognitive sciences, and one deeply related to model matching, is what makes two objects similar in the mind of a human being [64]. Four major psychological models of similarity are the following: geometric, feature-based, alignment-based, and transformational [65]. In geometric models, the entities are presented as points in N dimensional space, where a dimension in the space depends on user judgments and is discovered through interviews [104]. In feature-based models the similarity of entities is calculated by matching features of compared entities. Then, similarity is increased for similar features, and decreased for dissimilar features. In alignment-based models, entities are put in correspondence if they have more features in common, and features are put in correspondence if more entities have them in common [93]. In transformational models, the entities are transformed into each other by a series of transformations, and it is assumed that the similarity between object decreases as the number of transformations required to transform an entity to another entity increases. The transformational approaches are related to Kolmogorov complexity theory, which states that the complexity of an object is proportional to the length of the shortest program that generates that object [88].

It is important to note that none of the mentioned psychological models of similarity is best in all cases. This is expected, since if a specific psychological model would have been the best in all cases, this would be reflected in having one best model matching variant.

In our approach to model matching, we directly support geometric and feature-

based models of similarity. As discussed in Chapter 3, our calculation algorithm allows users to group attributes of a metamodel element, based on their perceived semantic similarity, and the calculation of similarities for a group of attributes is performed by using a multidimensional search. Moreover, the model elements are considered more similar if their features are more similar. Alignment-based and transformational models can be emulated by a user defined function for calculating similarity, assigned to a metamodel element in the configuration file.

To test the hypotheses about the nature of similarity, by using a computer, one needs to formally represent knowledge. Ontologies are the most common way to do this within a computer. As already mentioned, an ontology represents a conceptualization of a domain (which is a subset of the entire corpus of knowledge). This definition is very similar to the definition of a metamodel, since a metamodel is also a conceptualization of a domain. Moreover, an ontology can also be ”instantiated”, and an instance of an ontology is analogous to a model.

However, the ontology is not a metamodel. The main difference between an ontology and a metamodel is the intended usage. Where a metamodel has a *constructive* nature and is used to *construct* models, an ontology has a *descriptive* nature and is used to *reason* about reality. For this reason, the relations between metamodel elements, and the relations between ontology elements are of a different nature. For example, the relation named ”equals”, which denotes that one element in an ontology is equivalent to another element of that ontology, does not exist in metamodels. Moreover, ontologies usually have an ”open-world” semantics, while models usually have a ”closed-world” semantics.

The experiments related to the nature of similarity, and thus potentially usable in model matching, are performed in the area of *ontology matching*. Ontology matching deals with matching two or more ontologies in order to discover pairs of entities in both ontologies that represent the same concepts. Based on previous discussion, it should be clear that *ontology matching* is analogous to metamodel matching and is not analogous to model matching. One more confirmation for this can be found in the fact that virtually all approaches to ontology matching use *names* of entities to match them. However, while metamodel elements are usually named, elements of models are not. Thus, it can be concluded that ontology matching is not directly applicable to model matching.

Nevertheless, although ontology matching is not directly applicable to model matching, a class of approaches to ontology matching, so-called *instance matching*, can be used in model matching. In instance matching, the matching of ontologies is done by matching ontology instances. Since ontology instances correspond to models, the research done in this area may be useful in model matching. The basic idea in instance matching is to relate the similarity between instances of concepts to similarity between concepts. There is a large number of possible metrics used for this purpose; we will mention a few of them. The common metric used for this purpose is the *Jaccard similarity* [40, 74]. This metric calculates the similarity between concepts represented by instance sets A and B, as the ratio between the size of the intersection of sets A and B, divided by the size of the union of those two sets. Another commonly used metric in this context is the *K-statistic* [73]. This metric is calculated based on the intersection, union, and difference of sets A and B. Another metrics used in this context is (slightly adapted) *Pointwise Mutual Information* (PMI) [74]. PMI is a metrics used in statistics to measure the association between two random variables, given two outcomes of those variables. In instance matching, the random nature of the variables is approximated by using co-occurrence counts. The last metric we mention is the *Information Gain* (IG) metric [74]. IG describes the amount of information about some hypotheses that can be gained by observing the results of the experiments. In instance matching, IG is used to describe the difficulty of assigning an instance to a concept if that instance had already been assigned to another concept.

The main problem with all approaches to instance matching, in relation to model matching, is that they are not designed towards distinguishing between non-identical elements in ontology instances. This, however, is not the case with model matching, since in the case of model matching, structurally identical model elements are easily matched, but the problem is in discovering structurally non-identical model elements, that represent the same entity in matched models.

However, one techniques from ontology instance matching is used in our matching algorithm. In particular, we use (adapted) Jaccard similarity to calculate the similarity of two model elements considering their sub-elements.

Schema matching and relation to databases

A database schema is a collection of statements, in some formal language, that specifies a set of possible database instances [77]. This view of a database schema is very similar to the view of a metamodel as a collection of statements, that specifies a set of possible models‘[76]. Database schema specifies the tables in a database, the columns of those tables, the relations between those tables, and constraints on those tables and their parts. This is very similar to a metamodel where the tables correspond to metamodel elements, the columns correspond to metamodel element attributes, and associations between metamodel elements correspond to relations between database tables. The differences between metamodels and database schemas stem from their semantics. Thus, while metamodels use notions from object-oriented paradigms, like hierarchies or inheritance, and use an object-oriented semantics, the database schemas use notions from a relational model [56], like relations or domains, and use a semantics of relational algebras.

Within the database community there is a large corpus of work on *schema matching*. While this work is not directly applicable to model matching (since database schemas correspond to metamodels and not models), some ideas from this area can be applied to model matching. In particular, *instance matching* is a technique in schema matching that can be applied to model matching. Instance matching in databases is very similar to instance matching in ontologies. However, instance matching for databases actively takes into consideration the existence of imprecise data, and actively copes with similar, and not only identical, values of attributes. For example, in [58], the authors propose two approaches to schema matching using instance matching. One proposed approach is a constraint-based matching, where a set of constraints is extracted from the attributes of two rows being compared, and the similarity is calculated based on the extracted constraints. Another approach proposed is a content-based matching, where the similarity of the entities is calculated by performing a pair-wise comparison of instance values for different attributes of entities. The final similarity between tables is obtained by aggregating the obtained similarity values.

In our approach to model matching we use an adapted variant of content-based matching, for calculating similarities between attributes of model elements, in

case that those attributes are not grouped.

A research area in the database community more related to model matching is the area of *duplicate record detection* [57], also called *instance identification* [119]. Duplicate record detection deals with identifying matching records in two databases that have the same (or matching) schema. There are two major problems that need to be solved in duplicate record detection. One of them is matching the field values, and the other one is matching entire records.

Matching of field values is performed by comparing field values via a selected similarity metric. The classes of similarity metrics commonly used for this purpose are: character-based, token-based, and phonetic similarity measures. Character-based similarity metrics are geared towards matching strings that are typographically similar (i.e. these metrics are good for matching strings that have only typographical errors). These metrics can be used to match the values of string based attributes that have typing mistakes. Token-based similarity metrics are usable if the matched strings consist of tokens that should be matched separately. These metrics can be used to match the values of multi-word string based attributes, e.g. address fields. Phonetic similarity metrics are usable for matching strings if they are phonetically similar. These metrics can be used to match the values of string based attributes that model names, surnames, or other phonetically similar concepts.

Matching of field values directly corresponds to matching of attributes of model elements, in particular string-based attributes. In our approach, we support matching of attributes by allowing users to specify a similarity function for matching of an attribute. This metrics used in that function should correspond to the similarity metric (or a combination of metrics) that best fits the semantic of the attribute.

In order to match an entire record, the similarity results obtained by matching fields must be aggregated. The approaches to match records can be divided into two categories. In the first category are probabilistic and machine learning approaches that use training data to match records. If it is possible to specify the training data then these approaches are applicable to model matching. In the second category are the approaches that rely on domain knowledge, or on generic distance metrics to match records, without using the training data. Since the

training data may be hard to obtain, these approaches are more applicable to model matching than the approaches that use training data.

In our approach to model matching we do not support the probabilistic approaches, but we rely on generic, or user supplied threshold values to match model elements. However, it is plausible to use the training set (e.g. the benchmark data set discussed in Chapter 4) to train our model matching algorithm for setting the best threshold values for comparing models conforming to a particular metamodel.

Tree comparison

A tree is a (directed or an undirected) graph, where each two nodes are connected by a single path, i.e. a graph without cycles. In computer science many artifacts can be represented as directed trees, e.g. a directory structure. Therefore, a commonly encountered problem (in computer science) is comparing two artifacts that have a tree(-like) structure (e.g. directories, XML files, ...).

The result of the comparison is the difference between the trees representing these artifacts. This difference is commonly represented as a sequence of atomic operations of adding, deleting, inserting or replacing a node in the *initial* tree, in order to obtain the *target* tree [124]. If all these operations are assigned a cost, then the sum of all costs of all operations is denoted as the *edit distance* (of transforming an initial tree into a target tree). A *minimal edit distance* is an edit distance between two trees where the sum of costs of all involved operations is minimal [124].

In order to calculate an edit distance, nodes in the initial tree must be matched with nodes in the target tree. This process is called *tree matching*. In [124], an algorithm is given for calculating the *minimal edit distance* between a string and a labeled tree, and the authors prove that the problem of calculating the edit distance between two unordered labeled trees is in general NP-complete. This is caused by the lack of structural constraints in the trees being compared. However, if specific constraints are imposed upon the trees being compared, polynomial time algorithms for matching trees exist. For example, if a constraint is that only a complete subtree can be matched to a complete subtree, then

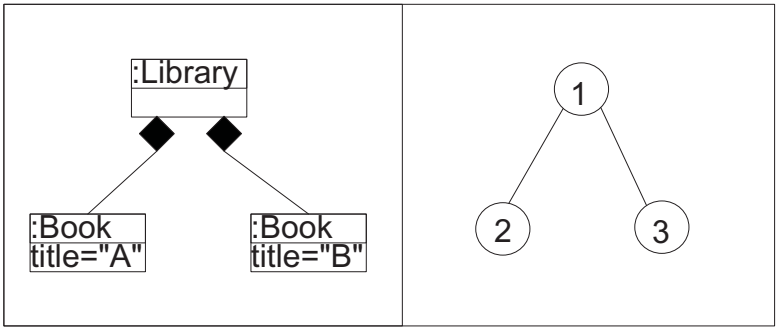

Figure 7.1: A model and a tree representation of the same model

polynomial time algorithms are given in [60, 90]. In case that the trees being matched have *a few internal nodes*, or *few branching nodes*, polynomial time algorithms achieved by using dynamic programming can be found in [68]. An algorithm applicable to *similar* trees (i.e. trees with small edit distance), working in $O(2.62^k \times poly(n))$ time and $O(n^2)$ space (where k is the edit distance and n is the size of the tree), is given in [38]. For ordered labeled trees a polynomial matching algorithm is given in [48] (*LaDiff* algorithm).

In general, models defined by using existing metametamodels (e.g. MOF, Ecore) can be represented as graphs. Moreover, if one considers model elements as nodes, and the instances of the aggregation relations (between model elements) as edges, then models can be considered as (unordered) trees. We will call a tree-like representation of a model: *a model-tree*. An example is given in Figure 7.1.

Thus, it seems plausible to use traditional *tree matching* approaches to match model elements and to compare models. However, the traditional approaches to *tree matching* rely on node labels as a means of matching nodes. But the labels of nodes representing model elements are not immediately distinguishable[1]. For example, in case all model elements have been assigned a universally unique identifier, this identifier can be used as a label, but the matching of elements in this case is trivial anyway. Another possible solution to this problem is to use

[1]Though the labels of metamodel elements are usually easily distinguishable and unique!

the signatures of model elements as labels, or to generate labels based on the features of model elements and their relations to other elements. However, even in the case that a label is defined for each model element, a direct application of the existing tree-matching algorithms to models is not possible. A reason for this is that each model element conforms to a metamodel element. Thus, it is not possible to match arbitrary elements (although they might have identical labels), but it is only possible to match elements that conform to the same metamodel element. This constraint, in theory, simplifies the algorithms for tree matching, and makes them available for model matching. One approach that considers this constraint is XDiff [36], that is developed for the matching of XML documents. However, the authors of XDiff impose a constraint that only elements that are of the same type [2], and whose ancestors are of the same type, can be matched. This constraint thus disallows matching of *moved* model elements. However, in models it is not possible to ignore the prospect of moved model elements, and in that case it can be proven that the problem of calculating *minimum edit distance* between two models is NP complete (e.g. by specifying a metamodel that allows for the representation of arbitrary graphs).

MDSE approaches

Recall that there are four identified *model matching* problem variants in MDSE [84]: static-identity based, signature-based, similarity-based, and language-specific. The application of these variants depends on syntactical and semantical constraints in models being matched. For example, if model elements carry universally unique identifiers (UUIDs), static-identity based matching can be performed. However, neither of the model matching variants is the best (or is even applicable) in all cases. For example, assuming that model elements are equipped with UUIDs, when a developer deletes a model element and later re-creates that same element, the match relation would fail to relate these two elements because the UUID of the new element has changed. Thus, an appropriate variant should be chosen depending on the syntactical and semantical aspects of the models being matched.

Model matching algorithms can be classified into two groups: metamodel-de-

[2] Model elements are of the same type if they conform to the same metamodel element.

pendent algorithms, and metamodel-independent algorithms. Metamodel-dependent algorithms utilize language-specific matching, i.e., they are geared towards a specific metamodel (e.g. UML). Metamodel-independent algorithms can be applied for comparing models conforming to arbitrary metamodels.

A well-known example of a metamodel-dependent approach is UMLDiff [123]. In UMLDiff the containment spanning-trees of both matched UML models are traversed and, at each level, elements that are ”the same” are identified. The traversal of the model-tree is top-down, and the elements that are significantly similar (i.e. whose similarity is above a certain threshold value) are matched.

As metamodel-independent approaches are concerned, an example differences calculation approach that relies on unique identifiers (UUIDs) for matching model elements is given in [39]. The problem with this approach, as well as all approaches that use only UUIDs for matching model elements, is that they work well when comparing evolutionary related models (i.e. when one model is an evolved version of another model). However, when comparing arbitrary models (or even models that have the same ancestor, but were developed in parallel), since their UUIDs are unrelated, this approach is not applicable.

Another metamodel-independent approach is SiDiff [79]. In SiDiff, the containment spanning-trees of both matched models are traversed, but a combination of bottom-up and top-down traversals are used. In particular, the bottom-up traversal is used until the match is found, and then a top-down traversal is used to match the children of two matched elements.

Finally, in DSMDiff [89], the containment spanning-trees of both matched models are traversed top-down, and at each level of the tree for all elements two metrics are calculated, and based on the values of these metrics the model elements are matched. The first calculated metric is the signature of nodes and edges, and the second calculated metric is the structural similarity of nodes. The signature of the node is calculated as a combination of the type, the kind, and the name of the node, The signature of the edge is calculated as a combination of the type, the kind and the name of the edge, and the signatures of nodes connected by that edge. In this approach it is possible to detect *moved* elements only between parents at the same level of the model tree (since only one top-down traversal of the

spanning tree is performed). Moreover, it is required that the signatures of model elements are identical, in order to match those elements. As a consequence, if the name of a model element in the initial model has changed just slightly in the target model, it would not be possible to match these elements. However, this is easily rectified by using a similarity-based matching for the signatures.

Our approach to model matching has some similarity to UMLDiff, but has the most similarity to SiDiff. In particular, we extended the configuration metamodel of SiDiff to allow our model matching algorithm to support all four recognized types of model matching. Moreover, unlike in SiDiff, we specify a representation mechanism for model differences, and we allow for moved model elements to be matched.

7.2.2 Metamodel and model co-evolution

Metamodel and model co-evolution is a term that denotes a coupled evolution of metamodels and models. A coupled evolution (co-evolution) of models occurs after a metamodel evolves. In this case it may be required to adapt (co-evolve) the models conforming to the *initial* version of the metamodel, such that they conform to the *target* (evolved) version, preserving the intended meaning of the initial model if possible. In order to perform this adaptation, a difference between initial and target metamodel must be discovered. Then, based on the discovered difference, adaptation of models can be performed. Approaches to model co-evolution can be classified as ”state-based”, if the difference between two metamodels needs to be calculated, and ”operation-based”, if a difference between two metamodels is supplied beforehand (e.g. is produced by a tool used to evolve the metamodel). In this dissertation we focused on ”state-based” approaches, that calculate the difference between two metamodels.

Similar to model comparison, a part of the process of calculating metamodel differences involves matching of two metamodels–*metamodel matching*. As already noted, the metamodels are similar to ontologies, and even more similar to database schemas, thus it seems plausible that methods for matching ontologies and schemas should be easily adaptable for matching metamodels. In the next two sections we will describe some existing approaches in ontology matching

and schema matching, and we will discuss the applicability of those approaches to metamodel matching. Finally, we will give an overview of the work on model co-evolution in the MDSE community, with an emphasis on metamodel matching.

Relation to ontology matching

There are two major classes of ontology matching approaches: approaches that match ontologies directly, and approaches that use explicit external knowledge to match ontologies [40]. Approaches that match ontologies directly can be further classified into terminological, structural, instance-based and global.

Terminological approaches to ontology matching are further classified into string-based matching approaches, and language-based matching approaches. String-based matchers rely on string-comparing functions to calculate the similarity between different terms. These methods are directly usable for metamodel matching, but their usability is restricted to renamed metamodel elements whose names are syntactically similar to the original names. Language-based matchers assume that the matched ontologies have given meaningful names to concepts they describe, and that those names are in the same language. If this is the case, the expressive and productive properties of natural languages can be used to match syntactically dissimilar, but semantically similar terms. The applicability of language-based matchers to metamodel matching is similar to the applicability of string-based matchers, since the metamodels matched are described in the same language, and, usually, meaningful names are assigned to metamodel elements.

In our approach, metamodel matching is done by transforming metamodels into models, and by performing matching on those models. Both string-based and language-based matchers can be used in metamodel matching, by defining a similarity function that takes two values of the name attribute of two metamodel elements, and returns a similarity between these values.

Structural approaches to ontology matching use the underlying graph structure of the ontologies to perform matching. The idea behind these approaches is that similar entities in one ontology will be related to similar entities in another

ontology. There are two classes of structural approaches. In the class of graph-based matchers, the entire underlying graph of an ontology is used for matching. In the class of hierarchy-based matchers, only certain relations are used to project an ontology into a tree. Examples of relations used are the ”part-of” relation and the ”child-of” relation. Since common relations used in metamodels are of similar nature to the relation used in ontologies, both the graph and hierarchical approaches to ontology matching are directly applicable to metamodel matching.

In our approach to metamodel matching, graph-based matching is directly supported. To achieve a hierarchy-based matching, zero weights should be assigned to certain relations between metamodel elements.

Instance-based approaches to ontology matching have already been discussed in the Section 7.2.1. These approaches are not usable for metamodel matching, since there are no models conforming to the evolved metamodel, which could be used to match on models conforming to the initial metamodel.

Global approaches to ontology matching combine the similarities obtained by applying several other approaches to ontology matching to obtain a match that is better than matches of any of the approaches applied individually. An example global approach is COMA++ [42].

In our approach to metamodel matching, a combination of approaches to calculate similarities can be only applied to match individual metamodel elements, but it is not possible to combine different approaches to match entire metamodels.

Approaches that use explicit external knowledge to match ontologies can be subclassified into approaches that rely on background-based similarity measures, and approaches that use a background ontology to match ontologies. Approaches that rely on background-based similarity measures utilize external lexical resources like thesauri, dictionaries, vocabularies, etc., to match ontology terms. One commonly used lexical database is WordNet [96]. Approaches that use a background ontology are performing matching of two ontologies through a third ontology (called background ontology), by relating terms in both matched ontologies to terms in the background, and by calculating their similarity based on those matches. While approaches that rely on external lexical resources can be directly used in metamodel matching, the approaches that use background on-

tologies cannot be used, since there is no third metamodel that can play a role of a background ontology.

In our approach to metamodel matching, approaches that use background-based similarity measures to calculate similarities between ontologies can be directly used by defining a similarity function that takes two values of the name attribute of two metamodel elements, and returns a similarity between these values

Relation to schema matching and schema evolution

A survey of approaches to schema matching [100], gives several, largely orthogonal, classifications, depicted in Figure 7.2.

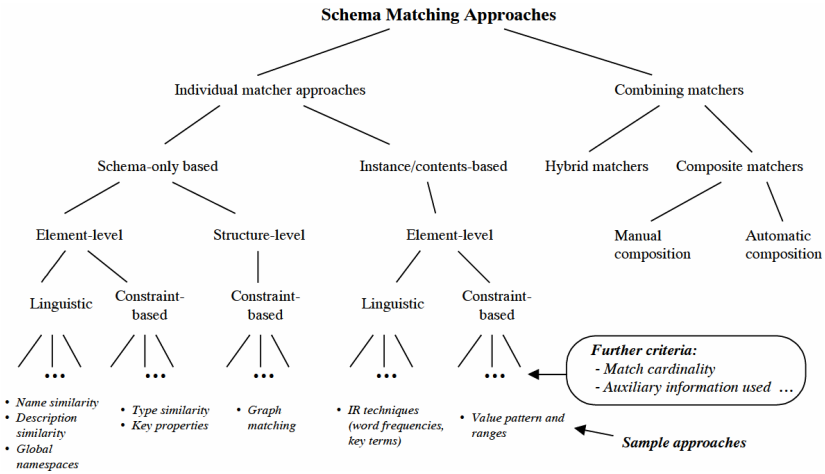

Figure 7.2: Classification of schema matching approaches

As already noted, instance-based approaches to schema matching are not directly applicable to metamodel matching, since instances of the evolved metamodel do not exist, but should be generated. Schema-only based matchers are directly applicable to metamodel matching [92]. In particular, element-level matchers can be effectively used to match attributes, while structure-level matchers can be used to effectively match model elements, and relations between model elements. Combined approaches are also directly applicable to metamodel match-

ing.

MDSE approaches

In this section, we will describe approaches to metamodel and model co-evolution in the field of Model Driven Software Engineering.

One of the first papers to cover the metamodel and model co-evolution topic was the paper by Favre [59]. In that paper Favre discusses the idea and role of co-evolution in the modeling process. The first paper to cover metamodel and model co-evolution, and to provide in-depth implementation details, was [106]. In that paper, the authors discuss the co-evolution process through the prism of domain-specific visual languages (DSVL). The authors discuss the influence of syntax, static semantics, and dynamic semantics on the co-evolution process. The authors represent metamodel differences in term of transformation operations, and also represent the co-evolution process as a transformation. Thus, this approach to metamodel and model co-evolution is ”operation-based”, and does not include model matching. Another ”operation-based” approach to model co-evolution is described in [118]. The author defines a set of operations that can be used for metamodel refactoring and evolution, and discusses the influence of those operations on models that should be co-evolved. This idea was adopted, and extended, in COPE [71]. The authors of COPE have performed a study to discover an extensive list of metamodel operations that are commonly used while editing metamodels [72]. All possible metamodel operations have been split into two subsets. In the first subset are the operations for which the syntactic and semantic influence on co-evolving models is known. For these operations it is possible to automate the co-evolution process completely. An example of this kind of operation is an operation that changes the name of a metamodel element. In this case, the co-evolution is trivial, since the change of the name of a metamodel element does not have any influence on models. In the second subset are the operations for which the syntactic and semantic influence on co-evolving models is not known. For these operations a manual intervention is needed to co-evolve the models. The latest advance in metamodel and model co-evolution is the proposal to use in-place transformations to perform co-evolution [95, 122]. The approaches that use in-place transformations do not calculate the differences between two metamodels, but merge those metamodels, which opens the possi-

bility to use in-place transformations to co-evolve models. There are two existing approaches to merging of metamodels. In the operation-based approach applied in [95], the difference between initial and target metamodel are given as a sequence of operations. Operations are grouped based on their influence of co-evolving models, and a merged metamodel is produced for each group of operations. In [122], the authors apply a state-based approach, and propose an algorithm that uses simple name-based string matching to match classes, attributes and references. Based on these matches, a merged metamodel is created and used as a basis for defining in-place operations.

A metamodel and model co-evolution approach that assumes ”state-based” differences is developed by Garcés et. al. [63]. In their approach, a heuristic metamodel comparison algorithm is used to compare two metamodels. The differences between two metamodels are described by using a differences metamodel. Based on the differences found, the co-evolution is performed by first executing a higher-order transformation, that generates a transformation that can be used to adapt all models conforming to the initial metamodel, and then by transforming all the required models by using the generated transformation.

Our approach to model co-evolution is ”state-based”. However, we do not define a special algorithm for the calculation of metamodel differences, but we transform metamodels into models, and use a model comparison algorithm to calculate metamodel differences. In this way, all possible atomic differences between metamodels are known, and, in our approach, are used as a basis for a co-evolution algorithm. Similarly to the approaches that use in-place transformations, the parts of the co-evolved models that conform to unchanged metamodel elements are kept the same, and only the parts of the co-evolved models that conform to the changed metamodel elements are changed. However, we do not require that the co-evolved models conform to initial, target metamodel, or any other metamodel, during the co-evolution process.

7.3 Future work

In Chapters 2, 4, 5, and 6, the future work related to topics discussed in those chapters has already been addressed, and in this section we will only recapitulate

those discussions. However, the future work presented in this section will not be organized by chapters, but according to some common criteria. We selected the most interesting criteria, which, sorted by ascending order of complexity, are: adapting our approaches to other formalisms, adding support for "3-way" model comparison, and including the semantics of models in our approaches.

Our approaches to the representation, calculation and visualization of model differences, discussed in Chapters 2, 3, 4, and 5 as well as our approach to co-evolution of models, discussed in Chapter 6, have been described by using the metametamodel defined by us. It would be interesting to examine the limits of our approaches in case they are applied for models that are based on more popular metametamodels such as Ecore (or MOF). The reason is that those metametamodels have more mature tools for create and editing models and metamodels, as well as more concepts that can be used to assist the users in the modeling process. The amount of research required for this extension is expected to be relatively low. The reasons for this are the following. The first reason is that our model differences representation mechanism is similar to the one used in Ecore, and thus it should be easy to adapt our representation mechanism to Ecore. The second reason is that our calculation algorithm is based on the tree-like representation of models, which is also supported by other metametamodels. Thus, this algorithm should be almost immediately applicable to models based on other metametamodels. The next reason is that our method for visualization of model differences works by mapping metamodel elements to graphical elements, and by relating metrics to metamodel elements. The metamodel elements in this case do not have to come from our metametamodel. Finally, e.g. in Ecore, it is possible to bijectively transform metamodels to models conforming to a metamodel for metamodels, thus opening the possibility of applying our approach to model co-evolution.

Our approaches to the calculation and visualization of model differences, discussed in Chapter 3 and 5, are defined to support only "2-way" model differences. However, in order to support the merge operation in model configuration management systems, these approaches should be extended to support "3-way" model differences[3]. This research track is expected to be more complex, since,

[3]Notice that the our model differences representation mechanism can be used for both "2-way" and "3-way" model differences.

as already mentioned in the introduction, this entire dissertation considers only "2-way" differences, and similar amount of work is expected for "3-way" differences. The simple solution of representing "3-way" difference by a pair of "2-way" differences does not work, since the "3-way" difference between two models may contain some atomic differences which are not contained in the "2-way" differences.

Our approach to the calculation of model differences, discussed in Chapter 3, and our approach to the co-evolution of models, discussed in Chapter 6, exploits only the syntax of models in their fully automated parts. It would be interesting to examine the possibilities of including the semantic of models in the fully automated parts of those approaches. Furthermore, our approach to generating metamodels and models, discussed in Chapter 4, also takes into consideration only on the syntactic structure of metamodels and models, and it would be interesting to explore the inclusion of the semantics in the generation process. This is the most complex research track, and in order to address it properly an entire research team might have to be involved. One of the reasons is that there is not yet consensus on the dynamic semantics of models in general (for static semantics, OCL [23] is taking the dominant role). Another reason is that even if the traditional techniques for describing semantics (e.g. structural operational semantics) would be applied to models, there are still many possible equivalences between labeled transition systems that arise from the models. For example, some equivalences are: bisimulation equivalence, trace equivalence, ready-trace equivalence, fault equivalence, etc. The problem is that those equivalences are applicable from case to case (i.e. from metamodel to metamodel), and it might be the case that a specific metamodel requires the definition of its own equivalence. Thus, the inclusion of the semantics in model comparison, in a generic manner, would imply that many of the important problems in the area of semantics of the modeling languages have been solved, thus making this research very important.

7.4 Final remarks

Model Driven Software Engineering is slowly, but steadily, replacing traditional Software Engineering as a preferred paradigm for developing complex software systems. However, some of the core support assets of MDSE, like models or

model transformations, are stil not mature enough regarding their formal syntax and semantics, and thus they may be interpreted in an ad-hoc way. This creates three problems.

The first problem is that the results published by one team of scientists are very hard, or even impossible, to assess and reproduce by other scientists. This problem can be somewhat alleviated by embodying the results of the research in a tool [116]. For instance, Ecore metametamodel is supported by a set of Java classes, that can be examined to discover parts of Ecore syntax and semantics that are not visible in the metametamodel itself.

The second problem is that the exchange of models and model transformations between different tools is very hard to accomplish. This problem occurs because the tools cannot exchange models (or model transformations) automatically, unless the syntax and semantics of models is formally specified. A step in the direction of the solution to this problem is the definition of modeling frameworks — if all models are defined by using the same metametamodel, then it is easier to exchange models and model transformations. However, as discussed in Chapter 2 of this dissertation, existing modeling frameworks are under-specified, and should be specified more precisely.

The third problem is that it is hard to define the syntax and semantics of model differences if the syntax of models is not fully formally specified. We addressed this problem in Chapter 3 of this dissertation. Our solution to this problem was to introduce a new modeling framework, and within it a new metametamodel. The introduced framework specifies the syntax, semantics, and relations between models, metamodels, in rigorous detail. However, the introduced framework is not meant to replace the existing frameworks, but it is rather focused on details of formal syntactic description of models and metamodels. Thus, while this framework can be used to define metamodels and models, and discuss their characteristics, it should be extended with more modeling concepts (like packages, operations, etc.), in order to be considered as a proper modeling framework. Moreover, we only developed tools to support model management (i.e. model comparison and co-evolution) for this framework, and did not develop tools for model development. Thus, if this framework is to be used in an industrial context, tools for model development should be developed. Furthermore,

tools for model comparison and model co-evolution we developed are academic, and should be refactored, if they are to be used in an industrial context.

From the above discussion it should be clear that, while the focus of this dissertation are methods for the support of model evolution (and co-evolution), in order to define these methods we had to solve other, more fundamental, MDSE problems. Thus, we conclude by arguing that both our ideas and methods related to model evolution, and ideas and methods related to modeling frameworks would be invaluable, and should be considered, in the process of design and implementation of any new modeling framework, or in the process of evolution of existing modeling frameworks.

Bibliography

[1] ATL transformation language. http://www.eclipse.org/atl/ (Viewed May 2011), .

[2] ATLAN metamodel zoo. http://www.emn.fr/z-info/atlanmod/index.php/Ecore (Viewed January 2011), .

[3] ATL transformations zoo. http://www.eclipse.org/m2m/atl/atlTransformations/ (Viewed May 2011), .

[4] A benchmark set of experimental data for assesing the quality of model-comparison tools. http://www.win.tue.nl/~zprotic/benchmark.html (Viewed May 2011).

[5] CIF: The compositional interchange format for hybrid systems. http://se.wtb.tue.nl/sewiki/cif/start (Viewed May 2011).

[6] CVS project. http://www.nongnu.org/cvs (Viewed May 2011).

[7] Cumulative frequency analysis with probability distributions. http://www.waterlog.info/cumfreq.htm (Viewed May 2011).

[8] The DOT language. http://www.graphviz.org/content/dot-language (Viewed May 2011).

[9] EMF compare project. http://www.eclipse.org/emf/compare/ (Viewed May 2011).

[10] Eclipse. www.eclipse.org (Viewed May 2011).

[11] Ecore. download.eclipse.org/modeling/emf/emf/javadoc/2.5.0/org/eclipse/emf/ecore/package-summary.html#details (Viewed May 2011).

[12] Enterprise Architect. http://www.sparxsystems.com.au/ (Viewed May 2011).

[13] Epsilon Comparison Language. http://www.eclipse.org/gmt/epsilon/doc/ecl/ (Viewed May 2011).

[14] EuGENia. www.eclipse.org/gmt/epsilon/doc/articles/eugenia-gmf-tutorial (Viewed May 2011).

[15] Fujaba. http://www.fujaba.de (Viewed May 2011).

[16] Git version control system. http://git-scm.com/ (Viewed May 2011).

[17] GME: Generic Modeling Enviroment. www.isis.vanderbilt.edu/projects/gme (Viewed May 2011).

[18] Gmf. www.eclipse.org/modeling/gmf (Viewed May 2011).

[19] OMG Model Driven Architecture. http://www.omg.org/mda/ (Viewed May 2011).

[20] MetaObject facility. www.omg.org/mof (Viewed May 2011), .

[21] MetaObject facility 2.4 specification. http://www.omg.org/spec/MOF/2.4/Beta2/PDF (Viewed May 2011), .

[22] Mercurial source control management tool. http://mercurial.selenic.com/ (Viewed May 2011).

[23] Object Constraint Language. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL (Viewed May 2011).

[24] Object Management Group. http://www.omg.org/marketing/about-omg.htm (Viewed May 2011).

[25] Query-View-Transformation language. http://www.omg.org/spec/QVT/1.0/ (Viewed May 2011).

[26] Rational Rhapsody. http://www-01.ibm.com/software/awdtools/rhapsody/ (Viewed May 2011), .

[27] Rational Rose. http://www-01.ibm.com/software/awdtools/developer/rose/ (Viewed May 2011), .

[28] Subversion software projec. http://subversion.tigris.org/ (Viewed May 2011).

[29] Swebok. http://www.computer.org/portal/web/swebok (Viewed May 2011).

[30] Metamodel-assisted model comparison tool. www.win.tue.nl/~zprotic/mctool.html (Viewed May 2011), .

[31] Model co-evolution tool. www.win.tue.nl/~zprotic/coevol.html (Viewed May 2011), .

[32] UML infrastructure V2.4. http://www.omg.org/spec/UML/2.4/Infrastructure/Beta2/PDF (Viewed May 2011).

[33] VIsual Automated model TRAnsformations - viatra. http://www.eclipse.org/gmt/VIATRA2/ (Viewed May 2011).

[34] Xtext. http://www.eclipse.org/Xtext/ (Viewed May 2011).

[35] The Standish Group report. http://www.standishgroup.com/chaos/intro1.php (Viewed May 2011), 1995.

[36] *X-Diff: an effective change detection algorithm for XML documents*, 2003.

[37] Software project management practices - failure versus success. `http://www.crosstalkonline.org/storage/issue-archives/2004/200410/200410-0-Issue.pdf` (Viewed May 2011), 2004.

[38] Tatsuya Akutsu, Daiji Fukagawa, Atsuhiro Takasu, and Takeyuki Tamura. Exact algorithms for computing the tree edit distance between unordered trees. *Theor. Comput. Sci.*, 412:352–364, February 2011. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/j.tcs.2010.10.002.

[39] Marcus Alanen and Ivan Porres. Difference and union of models. TUCS Technical Report No 527, TUCS Turku Centre for Computer Science, 2003.

[40] Zharko Aleksovski. Using background knowledge in ontology matching. Technical report, Vrije Universiteit, Amsterdam, 2008.

[41] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998. ISSN 0004-5411. doi: doi.acm.org/10.1145/293347.293348.

[42] David Aumueller, Hong-Hai Do, Sabine Massmann, and Erhard Rahm. Schema and ontology matching with coma++. In *International Conference on Management of Data*, pages 906–908, 2005. doi: 10.1145/1066157.1066283.

[43] D. A. van Beek, P. Collins, D. E. Nadales, J.E. Rooda, and R. R. H. Schiffelers. New concepts in the abstract format of the compositional interchange format. In A. Giua, C. Mahuela, M. Silva, and J. Zaytoon, editors, *3rd IFAC Conference on Analysis and Design of Hybrid Systems*, pages 250–255, Zaragoza, Spain, 2009.

[44] Edward H. Bersoff. Elements of software configuration management. *IEEE Trans. Software Eng.*, 10(1):79–87, 1984.

[45] Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20:10–19, April 1987. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/MC.1987.1663532.

[46] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional, aniversery (second) edition, 1995.

[47] Petra Brosch, Martina Seidl, and Gerti Kappel. A recommender for conflict resolution support in optimistic model versioning. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 43–50, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0240-1. doi: http://doi.acm.org/10.1145/1869542.1869549.

[48] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD International Conference on Management of data*, pages 493–504, New York, NY, USA, 1996. ACM. ISBN 0-89791-794-4. doi: doi.acm.org/10.1145/233269.233366.

[49] Antonio Cicchetti. *Difference Representation and Conflict Management in Model-Driven Engineering*. PhD thesis, Universita' degli Studi dell'Aquila, 2007.

[50] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, pages 165–185, 2007.

[51] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Meta-model differences for supporting model co-evolution. In *Proceedings of the 2nd Workshop on Model-Driven Software Evolution - MODSE2008*, Athene, Greece, 2008.

[52] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. Managing dependent changes in coupled evolution. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, pages 35–51, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02407-8. doi: http://dx.doi.org/10.1007/978-3-642-02408-5_4.

[53] Edmund H. Conrow and Patricia S. Shishido. Implementing risk management on software intensive projects. *IEEE Softw.*, 14:83–89, May 1997. ISSN 0740-7459. doi: 10.1109/52.589242.

[54] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45:621–645, July 2006. ISSN 0018-8670. doi: http://dx.doi.org/10.1147/sj.453.0621.

[55] Peter Domokosa and Daniel Varro. An open visualization framework for metamodel-based modeling languages. *Electronic Notes in Theoretical Computer Science*, pages 69–78, 2002. doi: doi:10.1016/S1571-0661(05)80531-6.

[56] F. Codd E. Derivability, redundancy and consistency of relations stored in large data banks. *SIGMOD Rec.*, 38:17–36, June 2009. ISSN 0163-5808. doi: http://doi.acm.org/10.1145/1558334.1558336.

[57] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 19:1–16, January 2007. ISSN 1041-4347. doi: http://dx.doi.org/10.1109/TKDE.2007.9.

[58] Daniel Engmann and Sabine Mamann. Instance matching with coma. In *Datenbanksysteme in Bro, Technik und Wissenschaft(German Database Conference)*, pages 28–37, 2007.

[59] Jean-Marie Favre. Meta-Model and Model Co-evolution within the 3D Software Space. In *Intl. Wshp on Evolution of Large-scale Industrial Software Applications at ICSM*, Amsterdam, September 2003.

[60] Pascal Ferraro and Christophe Godin. Optimal mappings with minimum number of connected components in tree-to-tree comparison problems. *J. Algorithms*, 48:385–406, September 2003. ISSN 0196-6774. doi: 10.1016/S0196-6774(03)00079-8.

[61] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977. ISSN 0098-3500. doi: doi.acm.org/10.1145/355744.355745.

[62] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. Practical adaptation of models to evolving metamodels. INRIA Technical Report No 6723, INRIA, 2008.

[63] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. Managing model adaptation by precise detection of metamodel changes. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 34–49, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02673-7. doi: http://dx.doi.org/10.1007/978-3-642-02674-4_4.

[64] R. L. Goldstone, D. L. Medin, and D. Gentner. Relational similarity and the nonindependence of features in similarity judgments. *Cognitive Psychology*, 23(2):222–262, 1991.

[65] Robert L. Goldstone and Ji Yun Son. Similarity. *Psychological Review*, 100:254–278, 2004.

[66] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *Int. J. Hum.-Comput. Stud.*, 43:907–928, December 1995. ISSN 1071-5819. doi: 10.1006/ijhc.1995.1081.

[67] Boris Gruschko, Dimitrios Kolovos, and Richard Paige. Towards synchronizing models with evolving metamodel. In *Proceedings of the 1st International Workshop on Model-Driven Software Evolution - MoDSE2007*, 2007.

[68] Magnús M. Halldórsson and Keisuke Tanaka. Approximation and special cases of common subtrees and editing distance. In *Proceedings of the 7th International Symposium on Algorithms and Computation*, ISAAC '96, pages 75–84, London, UK, 1996. Springer-Verlag. ISBN 3-540-62048-6.

[69] A. Henrich, H. W. Six, and P. Widmayer. The LSD tree: spatial access to multidimensional and non-point objects. In *VLDB '89: Proceedings of the 15th international conference on Very large data bases*, pages 45–53, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc. ISBN 1-55860-101-5.

[70] Markus Herrmannsdoerfer and Maximilian Koegel. Towards a generic operation recorder for model evolution. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP '10, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-960-2.

[71] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. COPE - automating coupled evolution of metamodels and models. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, pages 52–76, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03012-3. doi: http://dx.doi.org/10.1007/978-3-642-03013-0_4.

[72] Markus Herrmannsdoerfer, Sander D. Vermolen, and Guido Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *Proceedings of the Third international conference on Software language engineering*, SLE'10, pages 163–182, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19439-9.

[73] Ryutaro Ichise, Hiedeaki Takeda, and Shinichi Honiden. Integrating multiple internet directories by instance-based learning. In *Proceedings of the 18th international joint conference on Artificial intelligence*, pages 22–28, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[74] Antoine Isaac, Lourens van der Meij, Stefan Schlobach, and Shenghui Wang. An Empirical Study of Instance-Based Ontology Matching. pages 253–266. 2008. doi: 10.1007/978-3-540-76298-0_19.

[75] ISO - International Organization for Standardization. *International Standard ISO/IEC 25000 - Software engineering – Software product Quality Requirements and Evaluation (SQuaRE)*. Geneva, Switzerland, 2005.

[76] Ethan Jackson and Janos Sztipanovits. Formalizing the structural semantics of domain-specific modeling languages. *Software and Systems Modeling*, 8(4):451–478, 2009. ISSN 1619-1374. doi: 10.1007/s10270-008-0105-0.

[77] Manfred A. Jeusfeld and Uwe A. Johnen. An executable meta model for re-engineering of database schemas. In *Proceedings of the13th International Conference on the Entity-Relationship Approach*, ER '94, pages 533–547, London, UK, 1994. Springer-Verlag. ISBN 3-540-58786-1.

[78] Frédéric Jouault, Jean Bézivin, and Atlas Team. Km3: a DSL for metamodel specification. In *Proceedings of 8th FMOODS, LNCS 4037*, pages 171–185. Springer, 2006.

[79] Udo Kelter, Jürgen Wehren, and Jörg Niere. A generic difference algorithm for UML models. In Peter Liggesmeyer, Klaus Pohl, and Michael Goedicke, editors, *Software Engineering 2005*, volume 64 of *LNI*, pages 105–116. GI, 2005. ISBN 3-88579-393-8.

[80] S. Kent. Model Driven Engineering. In M. Butler, L. Petre, and K. Sere, editors, *Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM'02)*, volume 2335, pages 286–298, Turku, Finland, May 2002. Springer.

[81] Larry Klosterboer. *Implementing ITIL Configuration Management*. IBM Press, 1st edition, 2008. ISBN 0132425939, 9780132425933.

[82] Maximilian Koegel, Jonas Helming, and Stephan Seyboth. Operation-based conflict detection and resolution. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, CVSM '09, pages 43–48, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3714-6. doi: http://dx.doi.org/10.1109/CVSM.2009.5071721.

[83] Dimitrios S. Kolovos. Establishing correspondences between models with the epsilon comparison language. In *ECMDA-FA '09: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, pages 146–157, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02673-7. doi: http://dx.doi.org/10.1007/978-3-642-02674-4_11.

[84] Dimitrios S. Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F. Paige. Different models for model matching: An analysis of approaches to support model differencing. *ICSE Workshop on Comparison and Versioning of Software Models*, pages 1–6, 2009. doi: doi.ieeecomputersociety.org/10.1109/CVSM.2009.5071714.

[85] Patrick Könemann. Model-independent differences. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*

(CVSM '09), pages 37–42, Washington, DC, USA, 2009. IEEE Computer Society. doi: dx.doi.org/10.1109/CVSM.2009.5071720.

[86] René Krikhaar, Wim Mosterman, Niels Veerman, and Chris Verhoef. Enabling system evolution through configuration management on the hardware-software boundary. *Syst. Eng.*, 12:233–264, August 2009. ISSN 1098-1241. doi: 10.1002/sys.v12:3.

[87] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29(9): 782–795, 2003. ISSN 0098-5589. doi: dx.doi.org/10.1109/TSE.2003.1232284.

[88] Ming Li and Paul M.B. Vitnyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Publishing Company, Incorporated, 3 edition, 2008. ISBN 0387339981, 9780387339986.

[89] Yuehua Lin, Jeff Gray, and Frederic Jouault. DSMDiff: a differentiation tool for domain-specific models. *European Journal of Information Systems*, 16(4):349–361, August 2007. doi: 10.1057/palgrave.ejis.3000685.

[90] Shaoming Liu and Eiichi Tanaka. On the editing distance between unordered labeled trees. In *Electronics and Communications in Japan*, pages 1358–1371. Scripta Technica, Inc, 1996.

[91] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic schema matching with Cupid. In *Proceedings of the 27th International Conference on Very Large Data Bases - VLDB '01*, pages 49–58, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-804-4.

[92] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic schema matching with cupid. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 49–58, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-804-4.

[93] A. B. Markman and D. Gentner. The effects of alignability on memory. *Psychological Science*, 8:363–367, 1997. doi: 10.1111/j.1467-9280.1997.tb00426.x.

[94] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, March 2006. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2005.10.021.

[95] Bart Meyers, Manuel Wimmer, Antonio Cicchetti, and Jonathan Sprinkle. A generic in-place transformation-based approach to structured model co-evolution. *Electronic Communications of the European Association of Software Science and Technology (EASST)*, TBD 2011.

[96] George A. Miller. Wordnet: a lexical database for english. *Commun. ACM*, 38:39–41, November 1995. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/219717.219748.

[97] Alix Mougenot, Alexis Darrasse, Xavier Blanc, and Michèle Soria. Uniform random generation of huge metamodel instances. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 130–145, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02673-7. doi: http://dx.doi.org/10.1007/978-3-642-02674-4_10.

[98] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of UML diagrams. *SIGSOFT Software Engineering Notes*, 28(5):227–236, 2003. doi: http://doi.acm.org/10.1145/949952.940102.

[99] Hamilton Oliveira, Leonardo Murta, and Cláudia Werner. Odyssey-VCS: a flexible version control system for uml model elements. In *Proceedings of the 12th international workshop on Software configuration management*, SCM '05, pages 1–16, New York, NY, USA, 2005. ACM. ISBN 1-59593-310-7. doi: http://doi.acm.org/10.1145/1109128.1109129.

[100] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10:334–350, December 2001. ISSN 1066-8888. doi: http://dx.doi.org/10.1007/s007780100057.

[101] W. Kozaczynski S. Sendall. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, September 2003.

[102] Arne Schipper, Hauke Fuhrmann, and Reinhard von Hanxleden. Visual comparison of graphical models. In *ICECCS '09: Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 335–340, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3702-3. doi: dx.doi.org/10.1109/ICECCS.2009.15.

[103] D. C. Schmidt. Model-Driven Engineering. *Computer*, 39(2):25–31, February 2006.

[104] Roger Shepard. The analysis of proximities: Multidimensional scaling with an unknown distance function. i. *Psychometrika*, 27:125–140, 1962. ISSN 0033-3123. 10.1007/BF02289630.

[105] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, page 336, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7508-X.

[106] Jonathan Sprinkle and Gabor Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages and Computing*, 15:291–307, 2004. doi: 10.1016/j.jvlc.2004.01.006.

[107] Edward Suvanaphen and Jonathan C. Roberts. Textual difference visualization of multiple search results utilizing detail in context. In *TPCG '04: Proceedings of the Theory and Practice of Computer Graphics 2004 (TPCG'04)*, pages 2–8, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2137-1. doi: dx.doi.org/10.1109/TPCG.2004.33.

[108] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference computation of large models. In *ESEC-FSE '07: Proc. of the the 6th joint meeting of the ESEC and the ACM SIGSOFT symposium on The FSE*, pages 295–304, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4. doi: doi.acm.org/10.1145/1287624.1287665.

[109] Marcel van Amstel, Mark van den Brand, Zvezdan Protić, and Tom Verhoeff. Transforming process algebra models into UML state machines: Bridging a semantic gap? In *Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, ICMT '08,

pages 61–75. Springer-Verlag, 2008. ISBN 978-3-540-69926-2. doi: http://dx.doi.org/10.1007/978-3-540-69927-9_5.

[110] Mark van den Brand, Zvezdan Protić, and Tom Verhoeff. A unified framework for metamodel-independent model matching. Submitted to SoSyM special issue on Models and Evolution.

[111] Mark van den Brand, Zvezdan Protić, and Tom Verhoeff. Fine-grained metamodel-assisted model comparison. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP '10, pages 11–20, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-960-2. doi: http://doi.acm.org/10.1145/1826147.1826152.

[112] Mark van den Brand, Zvezdan Protić, and Tom Verhoeff. Generic tool for visualization of model differences. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP '10, pages 66–75, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-960-2. doi: http://doi.acm.org/10.1145/1826147.1826160.

[113] Mark van den Brand, Zvezdan Protić, and Tom Verhoeff. RCVDiff - a stand-alone tool for representation, calculation and visualization of model differences. *Proceedings of International Workshop on Models and Evolution - ME 2010*, 2010.

[114] Mark van den Brand, Albert Hofkamp, Tom Verhoeff, and Zvezdan Protić. Assessing the quality of model-comparison tools: a method and a benchmark data set. In *Proceedings of the 2nd International Workshop on Model Comparison in Practice*, IWMCP '11, pages 2–11, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0668-3. doi: http://doi.acm.org/10.1145/2000410.2000412.

[115] Mark van den Brand, Zvezdan Protić, and Tom Verhoeff. A generic solution for syntax-driven model co-evolution. In *Proceedings of the 49th international conference on Objects, models, components, patterns*, TOOLS'11. Springer-Verlag, 2011.

[116] Mark G.J. van den Brand and Kim Mens. Guest editors' introduction to the 3rd issue of experimental software and toolkits (est): A special issue

on academic software development tools and techniques (wasdett 2008). *Science of Computer Programming*, 75(4):214 – 215, 2010. ISSN 0167-6423. doi: DOI:10.1016/j.scico.2009.11.003. Experimental Software and Toolkits (EST 3): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT 2008).

[117] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35:26–36, June 2000. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/352029.352035.

[118] Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In Erik Ernst, editor, *Proceedings of the 21st European Conference on Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624. Springer-Verlag, July 2007.

[119] Y. Richard Wang and Stuart E. Madnick. The inter-database instance identification problem in integrating autonomous systems. In *Proceedings of the Fifth International Conference on Data Engineering*, pages 46–55, Washington, DC, USA, 1989. IEEE Computer Society. ISBN 0-8186-1915-5.

[120] Sven Wenzel. Scalable visualization of model differences. In *CVSM '08: Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models*, pages 41–46, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-045-6. doi: doi.acm.org/10.1145/1370152.1370163.

[121] Bernhard Westfechtel. A formal approach to three-way merging of EMF models. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP '10, pages 31–41, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-960-2. doi: http://doi.acm.org/10.1145/1826147.1826155.

[122] M. Wimmer, A. Kusel, J. Schnbck, W. Retschitzegger, W. Schwinger, and G. Kappel. On using Inplace Transformations for Model Co-evolution. In *Proceedings of the 2nd International Workshop on Model Transformation with ATL (MtATL 2010)*, 2003.

[123] Zhenchang Xing and Eleni Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 54–65, New York, NY, USA, 2005. ACM. ISBN 1-58113-993-4. doi: http://doi.acm.org/10.1145/1101908.1101919.

[124] Kaizhong Zhang, Rick Statman, and Dennis Shasha. On the editing distance between unordered labeled trees. *Inf. Process. Lett.*, 42(3):133–139, 1992. ISSN 0020-0190. doi: dx.doi.org/10.1016/0020-0190(92)90136-J.

[125] Steffen Zschaler, Dimitrios S. Kolovos, Nikolaos Drivalos, Richard F. Paige, and Awais Rashid. Domain-specific metamodelling languages for software language engineering. In *SLE '09: Proceedings of the international conference on software language engineering*, pages 334–353. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-12106-7. doi: doi.acm.org/10.1007/978-3-642-12107-4.

Appendix A

Multidimensional Search

In this section we will provide a detailed description of the similarity function for matching objects by considering only their attributes. However, we will not consider a problem of comparing two objects, but we will consider the problem of comparing all objects in one set of objects to all objects in another set of objects, because this variant of similarity calculation is used in our algorithm. Thus, the setup of the problem is: for all objects in a set of objects S_A of certain type, find all similar objects in a set S_B of objects of the same type, such that the similarity is calculated by using only the attributes of the objects. Each attribute type has an associated similarity function which returns a real value between 0 and 1, and each attribute type has an associated threshold value such that if the result of the similarity function is equal or larger than the specified threshold value, two objects are considered similar.

We will first introduce a naive solution and show its shortcomings. In a naive approach, a similarity between object B in a set S_B and object A in a set S_A is

calculated in $O(m)$, where m is the number of attributes of those objects. If there are n objects in set S_B, each of them needs to be compared with the object A, which takes $O(n)$ time. The maximum similarity between all found similarities is found in time complexity $O(1)$, thus the complexity of the algorithm is $O(m \times n)$. To calculate the similarities between all objects in set S_B and all (k) objects in set S_A, the total computational complexity is $O(n \times k \times m)$. For large n and k, although the complexity is polynomial, this operation becomes a bottleneck of the system.

One of the solutions to this problem is to consider the similarity calculation as a search problem and use the results from the field of multidimensional search. In order to solve this problem by using multidimensional search, the attributes of objects are to be considered as dimensions of a multidimensional space. The objects in the sets S_B and S_A are considered as points in the multidimensional search space, and for each point representing an object in the set S_A (search object) a search is performed to find the closest point in the search space from the set S_B (also called the nearest neighbor). The object represented by the nearest neighbor is considered the most similar object to the search object. In order to make the search faster than the naive approach, a search structure is created based on the elements of set S_B that allows for a faster search. There are many proposed structures and we will give a brief explanation of some of them. We will first describe approaches tuned to finding the exact nearest neighbor (NN algorithms). One of the simplest structures are k-d trees [61]. In this approach generalized binary trees are used to represent the elements of the set S_B. Another similar approach is introduced in [69], using LSD trees as a search structure. LSD trees provide different split strategy than k-d trees, and behave better for a wider range of data distributions. An approach presented by [108] facilitates the fast comparison between large models by using SSSV trees, which are based on LSD trees. The expected search time of these algorithms is $O(n * log(k) * m)$.

A somewhat different class of algorithms are algorithms for finding approximate nearest neighbor (ANN algorithms). These algorithms do not try to find the point that is the closest to the search point, but try to find the point that is possibly closest, within certain error margin. An approach by Arya et al. [41] solves the problem of finding the approximate nearest neighbor by using balanced-box decomposition (BBD) trees. The idea behind these trees is to separate the search

space into regions, such that each region can be associated to only one point from the search set. An advantage of this algorithm is that it is easily adapted such that more neighbors of a search point can be found fast.

However, both NN and ANN algorithms cannot be used directly in our calculation algorithm. One reason for this is that in order to use the NN (or ANN) algorithms, the values of the attributes must form a metric space. This means that, for example, if an object would contain an attribute of type *string*, the NN algorithms could not be used because strings in general do not form a metric space. Another reason why these algorithms cannot be used directly in our algorithm is that they are are tuned to geometric objects. In particular the nearest neighbor of a search point is the point that fits into a multidimensional ball (a ball of certain radius around a search point) of the smallest radius. However, this implies that the dimensions are related, and in the case that this is not true, these algorithms cannot be used. Consider for example the objects representing persons that have two attributes named *Age* and *DrivingLicenceNumber*. It is unclear what is the meaning of the multidimensional ball in the space defined by these attributes. Knowing this, it is clear that multidimensional search techniques must be adapted in order to be used as a multi-attribute search technique.

Next we will present our approach for comparing attributes.

We allow the users to separate all attributes into two classes. In the first class are attributes which type is (or can be cast to) a real number, or the attributes which can be transformed into real values such that those values form a metric space (for example characters can fit in this class). The attributes in the first class can be separated into groups of related attributes. For each group of related attributes, an ANN search will be used to find all the similar objects that are within specified threshold. We use ANN algorithms because they allow for finding multiple similar objects fast, which fits into the description of our similarity function, which returns true for all similar objects of an object. The group of an attribute is defined in the *ComparisonMMAttribute* instance that is related to the *MMAttribute* instance that the attribute is related to (the group is an integer equal or greater than 1). In the second class are all other attributes (the group of these attributes is set to 0). The second class contains attributes of types which do not form a metric space (for example, strings and arrays can be in the second class). Next, for

each group of attributes in the first class we calculate the similar objects. Then, for each attribute in the second class we calculate the similar objects (we provide default similarity functions for several types of attributes, for all other types the users should provide similarity functions). Finally, we use a logical expression which combines the obtained results. The default expression states that two objects are similar if all of their attributes are similar, but as already mentioned in Section 3.3 this expression can be redefined by domain experts to express the domain-specific way of calculating similarity between objects.

Appendix B

Types of mapping rules and example mappings

As already noted, the mapping between an arbitrary metamodel and a dot metamodel is represented by a set of rules. The goal of each rule is to provide a declarative description of the graphical shape which will be used to represent the instance of a specific metamodel element. Thus, each rule is related to one metamodel element, but is used to visually represent all model elements conforming to that metamodel element. There are five rule types. These types are formulated in such a way to enable an extremely wide range of visualization possibilities.

The detailed rationale behind all the rule types is the following: The *MMElements* are the main ingredients of the metamodels, thus they are included in most rule types. Attributes of *MMElements* are not mapped, because they are considered inseparable parts of *MMElements*, and are treated like that. Thus, all types of rules used to transform *MMElements* have an option to include the attributes in

one of the predefined ways (for example, attributes could be visualized in a rectangle which is connected to the node representing the mapped object, separated by horizontal lines and sorted by the name of the attribute). References are included in the mappings in three ways: First, while mapping an *MMElement* into an edge, two instances of *MMReference* of that *MMElement* are selected, such that the instances of *MElements* that are referenced by the instances of *MReferences* that are references by the selected *MMReferences* instances are chosen as the initial or target node connected by an edge. The second way of including the references in the mapping is represented by the fourth rule type and is self explanatory. The third way of including the references in the mapping is represented by the mapping of an *MMElement* into a *nexus*. In this mapping a set of references is chosen as the incoming edges in the *nexus*, and the set of references is chosen as the outgoing edges from the *nexus*.

Next we will give a detailed description of each rule type. Each rule expects a Metamodel element ID. A Metamodel element ID is needed in order to connect the rule to a specific metamodel element. The model elements conforming to the specified metamodel element will be visualized by using this rule.

B.1 Rule type 1

This type of a rule can be used to transform model elements to *dot* nodes. It can be used to represent classes in a class diagram, or states in a state machine. It has the following attributes:

- Metamodel element ID
- Node shape
- Attribute name that will be used as a Node label
- The format of visualization of metamodel element attributes
- Positioning attributes list

The *Node shape* is one of the possible shapes for *dot* nodes [8]. The attribute name is the name of the attribute that will be used as a label of this node. If the

metamodel element has no attributes, or the node does not need to have a label, this attribute can be set to the empty string. Positioning attributes is a list containing the identifiers of four attributes of a model element that will be interpreted as a positioning data for that element(top, left, bottom and right coordinates). The format of visualizations of model element attributes can be one of the following:

- NONE
- RECORD
- HIDDEN RECORD
- TREE

If the format is set to NONE, no attributes of the model element will be visualized. It is important to know that if the format is set to NONE, changes to attributes will not be visible in the visualization of the differences. If the format is set to RECORD, all the attributes are shown in a box, with each attribute separated from others by a horizontal or vertical line. The box representing attributes is connected to the node representing the model element. The HIDDEN RECORD format is similar to the RECORD format, the difference is that the edge that connects the attribute box with the node representing the model element is hidden. If the format is set to TREE, the attributes are visualized by using oval-shape nodes, which are connected to the node representing the mapped model element. An example of all four attributes representation formats is given in Figure B.1.

B.2 Rule type 2

This type of rule can be used to transform model elements to *dot* clusters. Clusters can contain other nodes, and are used to represent a containment-type hierarchical structures. This type of rule can be used to represent, for example, a complete class diagram, or a complete state-machine. It can be applied only to metamodel elements that contain other metamodel elements. It has the following attributes:

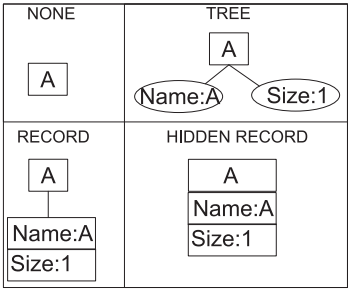

Figure B.1: Attributes representation formats

- Metamodel element ID
- Attribute name that will be used as a Node label
- The list of IDs of sub-metamodel elements that the selected metamodel element contains
- The format of visualization of model element attributes
- Positioning attributes list

Metamodel element ID connects this rule to a specific metamodel element. The attribute name is the name of the attribute that will be used as a label of the resulting cluster. The list of IDs of sub-metamodel elements is used to select which subobjects of the mapped model element will be visualized inside this cluster. All the subobjects (of a model element mapped to a cluster) that conform to the listed metamodel elements will be visualized inside the cluster. The format of visualizations of metamodel element attributes can be one of the following:

- NONE
- RECORD
- HIDDEN RECORD
- TREE

- INSIDE RECORD

The description of all of these formats is the same as in Rule type 1, except of the INSIDE RECORD, which denotes that the attributes should be visualized inside a cluster.

B.3 Rule type 3

This type of rule can be used to transform model elements to *dot* edges. It can be used to represent associations in a class diagram, or transitions in a state machine. It has the following attributes:

- Metamodel element ID
- From reference ID
- To reference ID
- From reference shape
- To reference shape

The rationale behind this element is the following: This type of rule should enable graphical representation of elements that have two or more references of the maximal cardinality 1. Thus, these model elements are actually connections between two or more model elements. For example, the transitions in a state machine or associations in class diagrams should be visualized by a Rule type 3. Thus, this model element will be visualized as a set of edges between all elements that are referenced by its reference having the same ID as the From reference ID, and between all elements that are referenced by its reference having the same ID as the To reference ID. The From reference shape and To reference shape, are shapes of the ends of created edges.

B.4 Rule type 4

This type of rule can be used to transform references to *dot* edges. It can be used to create tree-like structures, instead of cluster-like structures. For example, if one does not want to create a cluster out of a node, he can specify rules that map references of specific model elements to edges, thus creating a tree structure. This type of rule has the following attributes:

- Metamodel element ID
- Reference ID
- From shape
- To shape
- Edge line type

Reference ID is the ID of a reference whose instances will be mapped to edges. From and To shapes are the shapes of the edge begin and end. The *Edge line type* is the type of the line of an edge, and should be one of the recognized *dot* line types.

B.5 Rule type 5

This type of rule can be used to transform model elements to *dot* nexuses. Nexus is a special node that is connected to all instances of all referenced elements by the mapped model element. This node can be used to represent structures like choice, junction or join pseudo states in UML state machines. Specifically, this type of rules is good to represent the structures that connect more than two elements. It has the following attributes:

- Metamodel element ID
- Node shape
- Attribute name that will be used as a Node label

- The format of visualization of metamodel element attributes
- Positioning attributes list

The attributes of this rule type are the same as the ones in rule 1, and have the same meaning.

B.6 Examples

In this section we provide several examples of rules, and their application to the example model.

B.6.1 Example 1

In this example we specify rules that visualize a state machine model (and can be used for other models having the same metamodel) presented in a tree form in the middle of the Figure 5.4, to its appropriate graphical representation which is presented in the lower part of that Figure.

We will assume that the State metamodel element has an ID S1, that the Transition metamodel element has an ID T1, that the From reference of the Transition element has an ID T1R1 and that the To reference of the Transition element has an ID T1R2.

There are two rules required. The first rule is used to visualize states:

```
RULE:
Type: TYPE1
MetamodelElementID: S1
Shape: Circle
LabelAttribute: Name
AttributesVisualization: HIDDEN
```

This rule is of type 1, and it is thus related to the metamodel element with ID S1. It represents instances of that metamodel element as circles, the label inside the

circle is the value of the Name attribute of the element, and the element attributes are not explicitly visualized.

The second rule is used to visualize transitions:

```
RULE:
Type: TYPE3
MetamodelElementID: T1
FromReferenceID: T1R1
ToReferenceID: T1R2
FromReferenceShape: none
ToReferenceShape: normal
```

This rule is of type 3, thus all transitions will be turned into edges.

In the example, there is one transition, for which the From reference (the one with ID T1R1) references object with an ID 1, and the To reference (the one with ID T1R2) references object with an ID 2. Thus, this transition will be visualized as an edge between objects with ID 1, and ID 2, which are states in this example, and are visualized as circles.

Thus, these two rules can be used to visualize the basic state machines.

B.6.2 Example 2

In this example we will extend the metamodel presented in Figure 5.4 by creating a container for states and transitions (called StateMachine). The new metamodel, tree-view of an example model and appropriate visualization of the example model are presented in Figure B.2. In order to visualize the models conforming to the metamodel presented in Figure B.2, two rules defined in example 1 will be used, and one extra rule will be defined. We will assume that the StateMachine metamodel element has an ID SM0, and that it contains metamodel elements State and Transition. The extra rule is as following:

```
RULE:
Type: TYPE2
MetamodelElementID: SM0
```

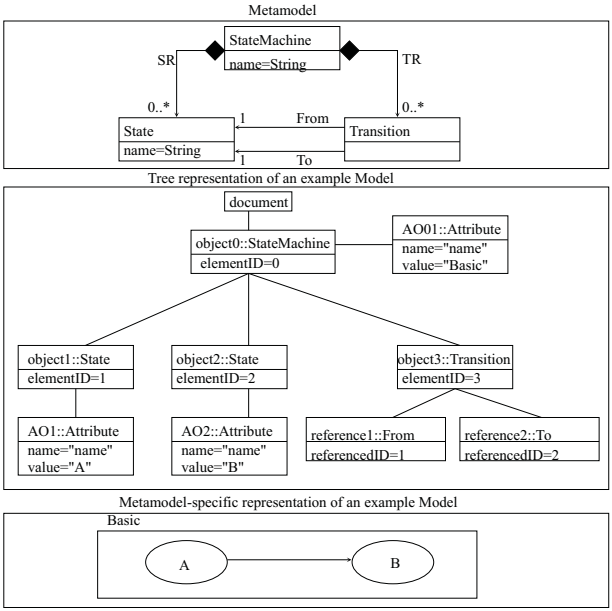

Figure B.2: Extended state-machine metamodel and an example model

```
SubelementsIDList: S1, S2
LabelAttribute: Name
AttributesVisualization: HIDDEN
```

This rule denotes that all state machines will be visualized as clusters, with their belonging states and transitions visualized inside of them. The attribute Name will be used as a name of the state machine, and no attributes will be visualized.

B.6.3 Example 3

In this example, we will further extend the metamodel presented in the example 2. We will extend this metamodel such that now a start state and an end state can be visualized separately. The extended metamodel, together with a tree-view of an example model, and an appropriate visual representation of an example model are presented in Figure B.3. In order to visualize models conforming to

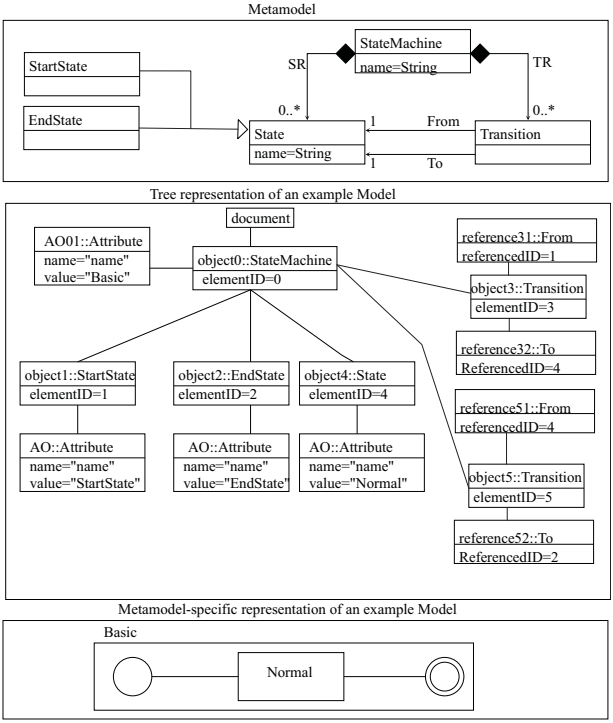

Figure B.3: Further extended state-machine metamodel and an example model

this metamodel, we will take three already defined rules, change one of those, and we will create two more rules. We will assume that the metamodel element StartState has an ID SS, and the metamodel element EndState has an ID ES. The first changed rule is the one for that is used for visualization of States, now visualizing states as boxes:

```
RULE:
Type: TYPE1
MetamodelElementID: S1
Shape: Box
LabelAttribute: Name
AttributesVisualization: HIDDEN
```

The new rules are as following:

```
RULE:
Type: TYPE1
MetamodelElementID: SS
Shape: Circle
LabelAttribute:
AttributesVisualization: HIDDEN

RULE:
Type: TYPE1
MetamodelElementID: ES
Shape: DoubleCircle
LabelAttribute:
AttributesVisualization: HIDDEN
```

Thus, an instance of a start state will be visualized as a circle, and an instance of an end state will be visualized as a double circle.

Appendix C

Possible metamodel differences

In this Appendix we describe all the possible types of atomic metamodel differences and (separated by a $\rightarrow$ symbol) the possible impact of those differences to the co-evolving models. This set of atomic differences is sound and complete. Notice that each type of metamodel difference is related to one group of metamodel differences introduced in Section 6.1. This relation is denoted by an abbreviation of a differences group (BRD, NBD, BSRD or BHRD).

1. In the *new* metamodel, an **element** was **deleted** (BRD) $\rightarrow$ The conforming model elements should be deleted from all the models.
2. In the *new* metamodel, an **element** was **added** (NBD) $\rightarrow$ Nothing should change in co-evolving models.
3. In the *new* metamodel, the **name** of an element was **changed** (NBD) $\rightarrow$ This does not have any influence on the conforming models, since the model elements are not related to the metamodel elements by name.
4. In the *new* metamodel, an **attribute** of an element was **deleted** (BRD) $\rightarrow$ The instance of that attribute should also be deleted from all model elements conforming

to that metamodel element.

5. In the *new* metamodel, an **attribute** was **added** to an element (BSRD) → The instance of added attribute should be added to all the model elements conforming to the changed metamodel element. However, a default value should be provided for all attributes added to a metamodel in order to obtain syntactic correctness of the resulting models. This default value can be provided in a static (per-metamodel) configuration file, thus this difference is *Breaking and semi-resolvable difference*.
6. In the *new* metamodel, an **attribute** of an element was **changed**; the following options are possible:
   (a) In the *new* metamodel, the **name** of the attribute was **changed** (NBD) → Nothing should be changed in the models, because models do not reference attributes by name.
   (b) In the *new* metamodel, the **type** of the attribute was **changed** (BSRD) → The values of that attribute in models might not be valid anymore. Thus, a transformation function that transforms the old values of the attributes to the new values of the new type should be provided in a pre-defined configuration file.
7. In the *new* metamodel, a **reference** of an element was **deleted** (BRD) → All instances of it should also be deleted from all of the model elements conforming to the changed metamodel element.
8. In the *new* metamodel, a **reference** was **added** to an element (BHRD) → The changes to model elements depend on the lower bound of the added reference. If the lower bound of the reference is zero (0), then, syntactically, the models are correct without any change. If the lower bound on the reference is not zero, then the appropriate instances of the reference should be added by a user, i.e. it is not possible to automatically infer a correct model.
9. In the *new* metamodel, a **reference** of an element was **changed**:
   (a) In the *new* metamodel, the **label** of the reference was **changed** (NBD) → Nothing should change in models.
   (b) In the *new* metamodel, the **bounds** of the reference were **changed** (BHRD) → A syntactic check should be invoked in the target model and appropriate warnings/errors should be issued in case the new bounds of the references are not respected in the model elements conforming to the changed metamodel element.
   (c) In the *new* metamodel, the **reference** was **changed** to refer to a different element (BHRD) → The reference instances do not point to the right type of elements, and a user should resolve the conflict.

10. In the *new* metamodel, a **contained element** was **deleted** (BRD) → All instances of the deleted subelement should be deleted from the instances of the model elements conforming to the changed metamodel element.
11. In the *new* metamodel, a **contained element** was **added** (NBD) → Nothing should change in models.

If in the *new* metamodel the contained element has been changed, then for each changed subelement the defined differences should be processed recursively.

Curriculum Vitae

Zvezdan Protić was born on 01.08.1978 in Novi Sad (Serbia). He finished primary school in the village of Tovariševo (Serbia) in 1993. He attended high school "Jovan Jovanović Zmaj" in Novi Sad (Serbia) from 1993 until 1997. Thereafter, he attended five-year graduate studies at Faculty of Technical Sciences in Novi Sad. He graduated in 2002, and the title of his graduation thesis was "Web-based system for knowledge assessment". His graduation title "Graduate Engineer of Computer Science", was equalized in 2007, according to the Bologna Declaration, to the title: "Graduate Engineer - Master". From 2002 until 2004 he was employed as an assistant helper at Faculty of Technical Sciences in Novi Sad. From 2004 until 2007 he was employed as a teaching and research assistant at Faculty of Technical Sciences in Novi Sad. From 2007 until 2011 he was employed as a Ph.D. candidate at Eindhoven University of Technology in Eindhoven.

Nederlandse samenvatting

Het traditionele software engineering paradigma is niet in staat om te gaan met de toenemende vraag naar software van hoge kwaliteit. Daarom is een nieuw paradigma, namelijk model-gedreven softwareontwikkeling (MDSE), snel om zich heen aan het grijpen.

MDSE belooft enkele van de problemen die traditionele softwareontwikkeling met zich meebrengt op te lossen door verhoging van het abstractieniveau. Daarom worden in MDSE modellen en modeltransformaties gebruikt om software te produceren in plaats van tekstuele programmacode, zoals dat in traditionele softwareontwikkeling het geval is. De modellen zijn veelal gebaseerd op grafen en worden ontwikkeld gebruikmakend van een grafische notatiewijze, met andere woorden, de modellen worden gerepresenteerd als diagrammen. Modeltransformaties kunnen worden gebruikt om verschillende soorten artefacten te produceren van modellen in allerlei stadia van een softwareontwikkelingsproces. Bijvoorbeeld, artefacten die gebruikt kunnen worden voor model checkers of simulatieprogrammas kunnen worden geproduceerd. Hierdoor kunnen software producten in een vroeg ontwikkelingsstadium geverifieerd of gesimuleerd worden, wat leidt tot een verdere reductie van de kans op fouten in het uiteindelijke

softwareproduct.

Echter, methoden en technieken ter ondersteuning van MDSE zijn nog steeds niet voldoende volwassen. Met name methoden en technieken voor configuratiebeheer van modellen zijn nog steeds in ontwikkeling, bovendien bestaat er nog geen generiek systeem hiervoor. In deze dissertatie, beschrijf ik mijn onderzoek dat gericht was op het ontwikkelen van methoden en technieken ter ondersteuning van configuratiebeheer voor modellen. Tijdens mijn onderzoek heb ik me met name gericht op het ontwikkelen van methoden en technieken voor het vergelijken van modellen en ter ondersteuning van model co-evolutie. De beschreven methoden en technieken zijn generiek en geschikt voor een toestandsgebaseerde aanpak voor configuratiebeheer van modellen.

Met betrekking tot het vergelijken van modellen heb ik methoden ontwikkeld voor het representeren, berekenen en visualiseren van toestandsgebaseerde model verschillen. In tegenstelling tot eerder gepubliceerd onderzoek, waar deze drie aspecten van modelverschillen in isolatie behandeld worden, zijn in mijn onderzoek al deze drie aspecten geïntegreerd.

Model co-evolutie is een term die het probleem beschrijft van het aanpassen van modellen in geval hun metamodel verandert. Mijn oplossing voor dit probleem bestaat uit drie stappen. In de eerste stap wordt een speciaal metamodel geïntroduceerd, namelijk een metamodel voor metamodellen. In tegenstelling tot traditionele aanpakken waar metamodellen als instanties van een meta-metamodel gerepresenteerd worden, worden metamodellen in mijn aanpak gerepresenteerd als modellen die instanties zijn van het metamodel voor metamodellen. In de tweede stap worden de eerdergenoemde methoden en technieken voor het vergelijken van modellen hergebruikt om metamodel verschillen te berekenen en te representeren. Dit is mogelijk omdat metamodellen gerepresenteerd worden als modellen. In de laatste stap definieer ik een algoritme that de berekende metamodelverschillen gebruikt om modellen aan te passen zodat ze voldoen aan het geëvolueerde metamodel.

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java - Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of*

Hybrid Systems. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical

Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Id-*

iomatic Crosscutting Concerns in Embedded Systems. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of As-*

pects. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering,

Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Med-*

ical Image Analysis. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented*

Discovery of Knowledge - Foundations, Implementations and Applications. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12