Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

A Solution for Concurrent Versioning of Metamodels and Models

Antonio Cicchetti[a] Federico Ciccozzi[a] Thomas Leveque[b]

a. Mälardalen University, Department of Innovation, Design, and Engineering (IDT), Mälardalens Högskola, 72123, P.O. Box 883, Västerås, Sweden, http://www.mdh.se

b. Orange Labs, Orange, 21 chemin du vieux chene, BP 98, 38 243 Meylan, France, http://www.orange.com

Abstract Model-Driven Engineering has been widely recognised as a powerful paradigm for shifting the focus of software development from coding to modelling in order to cope with the rising complexity of modern systems. Models become the main artefacts in the development process and therefore undergo evolutions performed in different ways until the final implementation is produced. Moreover, modelling languages are expected to evolve too and such evolutions have to be taken into account when dealing with model versioning. Since consistency between models and related metamodels is one of the pillars on which model-driven engineering relies, evolution of models and metamodels cannot be considered as independent events in a model versioning system.

This article exploits model comparison and merging mechanisms to provide a solution to the issues related to model versioning when considering metamodel and model manipulations as concurrent and even misaligned. A scenario-based description of the challenges arising from versioning of models is given and a running example is exploited to demonstrate the proposed solutions.

Keywords model-driven engineering, model versioning, model coevolution

1 Introduction

Nowadays, software systems pervade all the aspects of our everyday life, ranging from personal audio devices and mobile phones to air traffic control systems. As a consequence, complexity of software development is facing a continuous growth that requires several changes of the realisation approaches in order to reduce the intricacy of the development. In this respect, Model-Driven Engineering (MDE) has been proposed to facilitate the system development by creating, maintaining, and manipulating models, i.e., abstractions of a real phenomenon. These abstractions reduce the complexity of the problem by allowing developers to focus on the aspects that most matter in the design of the system, and permitting to reason about the scenario in terms of domain-specific concepts [Bez05]. As the model is an abstraction of the system in the reality, rules and constraints for building the model have to be properly stated through a corresponding language definition: a metamodel describes the set of available concepts and well-formedness rules a model must conform to[1]. Moreover, since models become first class citizens exploited for the development, a system is developed by refining models starting from higher and moving to lower levels of abstraction; refinement is implemented by transformations over models [Ken02]. A model transformation converts a source model to a target model preserving their conformance to the respective metamodels [CH06].

In order for MDE to be fully adopted in industrial settings, it is of paramount importance that developers are provided, at least, with the same level of facilities available in code-centric approaches. For instance, version control systems (VCSs) have been proven successful in code versioning, but they are only partially suitable for handling versioning in the modelling domain. In fact, differences and conflicts between versions of a same artefact are usually detected at file-level through line-oriented text comparison. However, even if taking into account model XMI serialisations, the mismatch between text and model levels of abstraction may lead to erroneous detection of differences and hence conflicts [AP03, KPP06]. Therefore, in the latest years a number of research works have been devoted to versioning models and metamodels at the appropriate level of abstraction, advancing the state of the art in (meta)model differencing, versioning, and related co-evolution problems.

Although specific techniques for model and metamodel version management exist, in general metamodel and model evolution issues are considered as independent from each other. However, real-life version management demands support of these evolutions in a concurrent way, since manipulation rates of models and metamodels are often different and not always aligned, disclosing additional problems [Fav05]. For instance, modifications made to the local version of a model could be operated in conformance to an older version of the metamodel currently stored in the shared repository, either because the developer did not update her current revision yet, or because she did not want to migrate to the newer version of the metamodel due to, e.g., tool availability, licenses, and so forth. We define such phenomena as *misalignment*s between metamodel and model versioning, since in theory each model existing in local repositories should be first migrated to conform to the newer version of the metamodel and then edited accordingly. In this way, all the local working copies would be re-aligned with the metamodel version committed to the shared repository, and each modification would make sense in the current metamodel revision.

This work introduces solutions for model versioning when considering metamodel and model manipulations as possibly concurrent and misaligned. As preliminarly discussed in [CCLP11], developers may opt to work with different versions of a metamodel. Therefore in this work we provide facilities to support the re-alignment of their changes with respect to the newest version of the metamodel stored in the shared repository. In particular, developers can choose to consider their manipulations *i*) valid in the previous version of the metamodel, *ii*) to migrate them in the new version of the metamodel, or *iii*) to try to apply them to the model revision resulting from the migration due to the metamodel update. It is worth noting that in

[1]Even though in general there are differences between the terms *metamodel* and *language*, in the scope of this article they will be used as synonyms

any case well-formedness issues arise, since neither intended modifications on an older version of the language could be completely valid in a newer version, nor manipulations operated by means of an older language could be legal in a more recent one. In this respect, the proposed mechanisms act directly on the modelling artefacts defined in their related language through model differencing and transformations as main instruments, and exploit (meta)models merging techniques in order to make manipulations compatible. Eventually, a subset of the available modifications is committed (updated) to the shared repository (local working copy) by selecting only those compatible with the target metamodel. In this respect, we propose an automated filtering operation able to distinguish the valid changes to be applied.

The remainder of this article is structured as follows. Section 2 presents motivations and contribution in relation to those aspects of the proposed approach that have already been partially explored in the current state of the art. In Section 3 we describe the note-worthiest scenarios of misaligned evolutions, while Section 4 illustrates corresponding challenges entailed by the language manipulations we consider. The proposed solution is introduced in Section 5 through a description of the general approach as well as the involved core artefacts and techniques. A detailed discussion of the techniques which the implementation relies on and the application of the process to a running example is given in Section 6. An evaluation of the current solution and the issues that remain still unsolved are presented in Section 7. Eventually, conclusions and possible future enhancements conclude the article in Section 8.

2 Background & Related Work

The need for adequate support to maintenance activities emerged as soon as software development reached the maturity to be employed in real-life applications. In fact, from the dawn evolution has been recognised as unavoidable to preserve and/or enhance user's satisfaction [Leh84]. Thus, the history of source code development proceeded together with more and more complex techniques to deal with the management of its evolution. In this respect, state of the art tools provide a repository, either centralised or distributed, in which the current version of the application is stored. Then, concurrent development may be controlled in a *pessimistic* way, meaning that each available artefact is locked whenever it is accessed, and released when saved back. In general, the efficacy of this solution decreases with the growth of the development environment due to the increased likelihood of collisions among access attempts.

In order to allow a more efficient process, in an *optimistic* approach each developer has a local workspace where modifications are done and later committed to the repository in order to update the current version and advertise it to the other developers. Moreover, whenever other developers want to *commit* their own changes, they will first need to *update* their local workspaces, *merge* their manipulations and the ones already committed, and then commit the resulting revision back to repository. Consequently, locking mechanisms can be avoided (even if still allowed when necessary) at the price of possible conflicts, that is divergences arising among overlapping modifications, that are typically fixed during the merging phase [ASW09].

The growing adoption of MDE in complex software development entailed the need of appropriate support for versioning management. Therefore, (meta)model version management has gained increasing attention within the current research in the MDE field. In the following, those efforts for achieving corresponding versioning solutions are described in order to better clarify the basic concepts and techniques that underpin the contribution of this article.

2.1 Model versioning

MDE promotes models as first class citizens exploited to generate code, perform analysis of system's properties, and so forth [Bez05]. Therefore, the need of model versioning support arises as soon as an MDE approach is adopted. Due to the relevant expertise coming from source code management, model versioning has been initially dealt with by relying on text-based mechanisms, such as Subversion [Tig] and CVS [CVS]. These are based on the concept of differences: in general, the latest version of the model is stored completely to be ready to use, whereas each of the past revisions is memorised in terms of differences with respect to its successor (or analogously predecessor). In this way the repository size can be optimised since usually the amount of changes is considerably smaller than the number of elements existing in the whole model.

In order to make models compatible with those techniques they are *serialised* in a more-or-less structured format ranging from plain text to XML documents. In this way, their versioning can be reduced to source code management, and comparison, conflict detection and resolution are operated line-by-line or through tree-based approaches. However, these solutions suffer an abstraction level mismatch, i.e., text-based techniques cannot grasp manipulations rationale at modelling level [KPP06]. For instance, serialisation procedures could produce textually different artefacts that however contain the same information [AP03]. Consequently, a number of techniques have been introduced for detecting, storing, and visualising [BP08] the evolution at modelling level. Approaches like the solutions proposed in [OWK03, XS05] are tailored for a certain modelling language as the UML; such restriction allows hypotheses to be made and constraints to be used for a better recognition of manipulations. Other mechanisms are language independent and can be applied to any kind of models [LGJ07, CDP07, RV08]; typically, these approaches have to face the intrinsic complexity of element matching [KDPP09].

Similarly to what illustrated for source code, models are edited locally and then committed to the repository. When updating the local workspace with the latest revision conflicts can occur that need to be solved; in this respect, dealing with models discloses a number of additional problems due to their intrinsic nature, as the issue of collisions at semantics level that are not detectable syntactically [AK09, CDP08].

There exist also mechanisms to document model histories by storing snapshots at corresponding points in time of the development life-cycle, that can hence be seen as an orthogonal representation of model evolution with respect to the differencing-based versioning [TDD00, Obj07]. For a more extensive survey on model versioning tools, related methodologies, and raising difficulties the reader is referred to [ASW09]; for the purpose of this article it is worth mentioning that the problem of model migrations due to metamodel adaptations is considered as a separate issue, or, in other words, there are no traces about model manipulations because of co-evolutionary side effects.

2.2 Metamodel versioning

The MDE vision prescribes the use of models in all aspects of the software development life cycle. In this respect, Domain-Specific Languages can be exploited to provide an abstraction level tailored to the domain experts by providing them with concepts closer to their expertise and to improve automation of derived artefacts [Bez05, KT08]. Due to their specificness, modelling languages are typically designed and subsequently maintained in an incremental manner; in this way, their expressiveness can be manipulated to add, fix, and/or refine available concepts [Fav05]. Versioning metamodels poses a twofold issue: metamodel evolution has to be detected and stored (as for models), and its side effects with respect to existing models have to be analysed and (possibly automatically) fixed. For the former problem, since metamodels are models themselves, the techniques dealing with model versioning can be transferred also to metamodelling settings [CDP09]. However, for the latter whenever a metamodel evolves, corresponding migrations have to be operated since existing model instances could not be well-formed any more [SK04]. This problem is known as co-evolution or co-adaptation, and a relevant number of research works propose solutions ranging from manual co-adaptation [RKPP10, SK04], to semi-automatic migration through re-use of recurring strategies [HBJ08, Wac07, DIP11] and automatic co-evolution directly generated from the manipulations made to the metamodel [CDP09, DV07].

Analogously to model versioning, metamodel evolution and the corresponding model co-evolution are managed as happening instantaneously, meaning that they are supposed to be operated at the same time and with no misalignment between metamodel revisions and model manipulations. In the following, we illustrate the problem of managing the concurrent modification of models and metamodels, that not necessarily happen at the same time and in a coordinated way. On the contrary, it is normal practice to edit models by means of, e.g., an older version of a modelling language in order to avoid licensing and/or tooling problems entailed with the migration to the latest version of the language [CCLP11].

3 Contribution

As discussed so far, model and metamodel versioning has been widely recognised as relevant problem demanding for adequate support in order to improve the adoption of MDE in industrial software development. Nonetheless, metamodel and model manipulations have been considered as happening in separate worlds and not overlapping each other; on the contrary, in this work we claim that they have to be considered as concurrently participating in version management. Since the contemporary revision of both languages and models carries further problems that have to be considered, the discussion contained in this article is based on the assumption that the language is explicitly versioned, meaning that in the repository both model and metamodel revisions are available. Moreover, due to memory and performance costs the evolution is stored in terms of differences between subsequent versions, whereas (as typically happens for source code) only the newest is kept as the (meta)model as a whole [CDP07].

In order to better clarify the mentioned evolution scenario, in Figure 1 it is depicted a general development situation in which at a first stage the metamodel $\mathtt{MM}_a$ is checked out through an update operation from the repository into the local workspace together with the current version of the model, i.e., $\mathtt{M}_a$[2].

Metamodel and model manipulations happen at different rates with respect to the time line of the development process: in fact, after a major stabilisation of the language in which metamodel refinements can be more frequent, model modifications

[2]For the sake of readability conformance relations have been omitted and they are given implicitly through the model name.

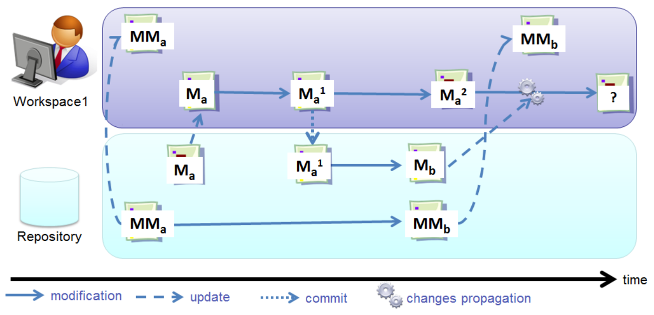


Figure 1 – Differences between metamodels and models evolution rates and arising issues

happen at more-or-less regular cadence due to the normal development and maintenance tasks, while metamodel adaptations are seldom operated in order to meet unforeseen requirements. For example, in Figure 1 $\mathtt{M}_a$ is modified in `Workspace1` and committed as the revision $\mathtt{M}_a^1$ into the repository.

The evolution pattern drawn so far can proceed smoothly until a metamodel adaptation happens: in fact, users could still be editing their models by means of the old version of the language in `Workspace1`, hence modifications can become inconsistent with respect to the current language version in the repository. For instance, in Figure 1 an evolution is operated on $\mathtt{MM}_a$ in order to obtain $\mathtt{MM}_b$: while $\mathtt{M}_a^1$ in the repository is migrated to $\mathtt{M}_b$ to restore its well-formedness with $\mathtt{MM}_b$, in the local workspaces there exist pending updates performed as conforming to $\mathtt{MM}_a$, i.e., $\mathtt{M}_a^2$ in `Workspace1`. As a consequence the following issues arise: should developers first try to migrate local models to $\mathtt{MM}_b$ and then merge the migrated manipulations with the revision in the repository? Or alternatively should they retrieve the latest version in the repository before the co-evolution, merge the local changes, and then migrate the result to $\mathtt{MM}_b$? For instance, while $\mathtt{M}_a^2$ has been obtained from the latest version before the migration "missed" modifications made within `Workspace1` and updating directly from $\mathtt{M}_b$ could cause loss of information.

At this point it is worth noting that, as explained in detail later on in the article, developers do not always work with the latest version of the used modelling language (and they will continue to use the old version of the language even when notified of available updates) due to licensing problems or tool chaining stability, to mention just a few. Of course, an ideal development process would entail a synchronised environment in which all the developers are using the same set of tools and the same version of each of them. Unfortunately, in industrial settings such assumption does not hold, not only among sites located in different regions of the world, but even across several departments placed in the same building.

In the following sections, we first illustrate in details a set of challenges that have to be faced when metamodels and model evolutions are considered as concurrent; then, in Section 5 we describe the solutions we propose to tackle these challenges.

4 Challenges

According to the MDE vision, models are expressed through a language which is formally defined in terms of a metamodel. In fact, as source code files are compatible

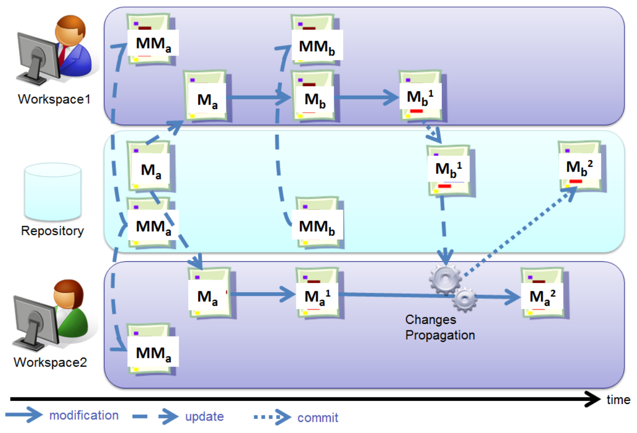


Figure 2 – Commit changes without metamodel update

only with a subset of corresponding language versions, a model is generally conforming to and compatible with a subset of metamodel versions. Managing possible evolutions of the language in terms of versions, and consequently models conforming to them, is a challenging task that is crucial in order to maintain a fully synchronised and consistent modelling environment especially, though not only, in case of distributed development.

In the following, we describe three common scenarios in which the challenges related to concurrent versioning usually arise:

- **Scenario 1:** Changes to a model that conforms to an old metamodel version are committed without updating metamodel version in the user workspace (see Figure 2);
- **Scenario 2:** Changes to a model conforming to an old metamodel version are committed and the metamodel version is updated in the user workspace (see Figure 3);
- **Scenario 3:** A model version designed according an old metamodel version is retrieved while metamodel version in the user workspace is not downgraded (see Figure 4).

In Scenario 1 (Figure 2), concurrent modifications of a model $\mathtt{M}_a$ are performed in conformance to metamodel $\mathtt{MM}_a$ starting from the same model version in both `Workspace1` and `Workspace2`. A new metamodel version $\mathtt{MM}_b$ is then created and updated to `Workspace1`; then, in the same workspace, $\mathtt{M}_a$ is co-adapted to conform to $\mathtt{MM}_b$ resulting in $\mathtt{M}_b$. $\mathtt{M}_b$ is modified and subsequently committed to the repository and such operation leads to the creation of version $\mathtt{M}_b^1$. Eventually, in `Workspace2` modifications to $\mathtt{M}_a$, resulting in $\mathtt{M}_a^1$, must be saved without updating metamodel version in `Workspace2` to the latest available in the repository (i.e., $\mathtt{MM}_b$) for two possible reasons: (i) forcing the usage of $\mathtt{MM}_b$ is costly since it implies updating of tooling related to the metamodel and time consuming since the current model version has to be migrated to the new metamodel version, and (ii) there could be tool licensing issues (e.g., in tool chains) for different users using different language versions. Moreover, consistency is supposed to be guaranteed among the version committed on the repository and

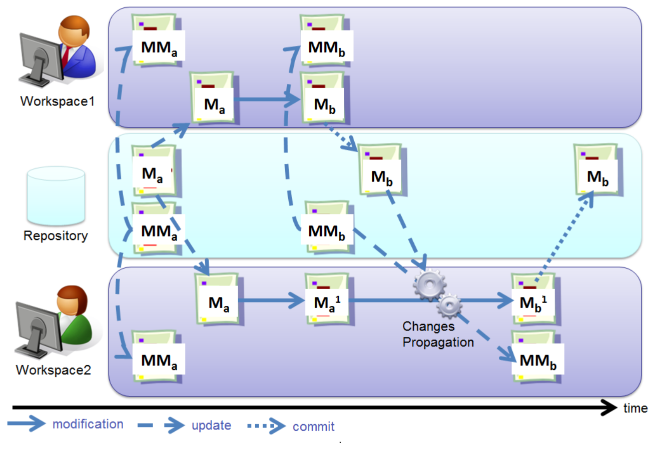


Figure 3 – Commit changes with metamodel update

the local revisions possibly conforming to different metamodels. A solution is needed for exchanging updates between workspaces, especially with different language versions, without losing information along the way from one language version to another. Committing to the repository is a complex versioning operation, involving model differencing and changes propagation among different versions of both metamodels and models, that has to be properly handled.

It is worth noting that the changes made on both $\mathtt{M}_a$ to obtain $\mathtt{M}_a^1$ and on $\mathtt{M}_b$ to obtain $\mathtt{M}_b^1$ conform to their corresponding metamodels, i.e., $\mathtt{MM}_a$ and $\mathtt{MM}_b$; in this respect, some modifications make sense for both the metamodels, while others are specific for either one or the other. Therefore modifications made on the repository may not be completely propagated to `Workspace1` and the other way around. In Scenario 2 (Figure 3), unlike Scenario 1, we assume that there are no issues preventing the update of metamodel version among workspaces and repository. Therefore, the first operation is to update the metamodel $\mathtt{MM}_a$ in the `Workspace2` to version $\mathtt{MM}_b$ and then changes operated to $\mathtt{M}_b$ in `Workspace1` and committed to the repository can be propagated to $\mathtt{M}_a^1$, in `Workspace2`, resulting in $\mathtt{M}_b^1$. Updating from repository to workspace is also a complex versioning operation involving model differencing and changes propagation among different versions of both metamodels and models that has to be properly handled. Versioning challenges do not only arise when going forward to newer versions of the metamodel. In fact, in Scenario 3, we depict how consistency may be undermined also when trying to go backwards to a previous version of a model which conforms to an older metamodel version, while keeping the current metamodel version in the workspace (Figure 4). Such operation may be needed when bugs in an old software version have to be fixed and the tooling related to the old language version is no longer available.

As the file-based, model VCSs save only differences between versions in order to reduce the repository size, retrieving an old model version is performed by composing all the related differences that led to the current revision. While file version differences are usually expressed as line differences, model version differences conform to a specific difference metamodel (see Section 5.2). As a result, model difference representations depend on the metamodel version taken into account. Therefore, retrieving

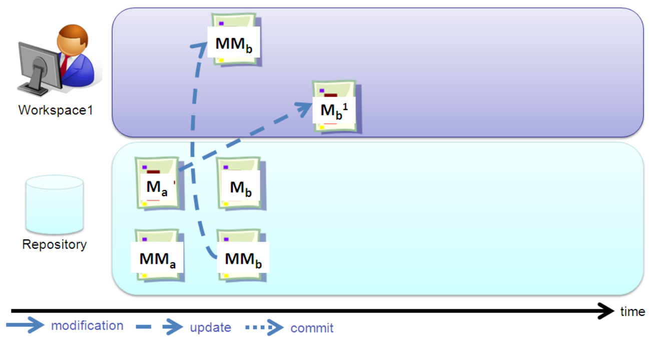


Figure 4 – Retrieve old version without its related metamodel version.

a model version may involve the composition of model evolutions expressed in different languages. In this respect, representing differences between two model versions conforming to different metamodels represents a non-trivial challenge for which we provide a possible solution in Section 5.2.

5 Proposed Solution

As discussed so far, challenges arise when managing the concurrent evolution of models and metamodels; therefore, let us suppose that in the repository there is the last version of the model as migrated by the co-evolution transformation. Moreover, in the workspaces there exist documents storing differences between the latest version updated from the repository and the current one. In this respect we propose to exploit model differences to support evolution storage: they are used for representing both metamodel and model manipulations.

5.1 Assumptions

Our approach is based on the following assumptions:

1. Model versions are stored in a shared repository and users work on a copy in isolation in their workspace. In this way we are able to optimise concurrent modifications and minimise loss in changes propagation;

2. Each model element is uniquely identified by a set of its attribute values and those of its container (recursively). Two elements are considered as versions of each other when they share a common information part. In order to distinguish if two elements are different versions of the same element or simply two different objects, we consider that two elements are versions of each other when the values assumed by their identifying attributes match;

3. A relationship is considered as a specific attribute type. Its value represents the destination model element identifiers;

4. A model always references to its corresponding metamodel. This ensures the ability to identify which is the corresponding metamodel for achieving a correct application of model comparison mechanisms;

5. There should be a root metamodelling level where the language is defined by means of itself in order to ensure the absence of infinite evolution levels.

5.2 Core Artefacts and Techniques

In this section we present a detailed description of the main modelling artefacts and techniques the proposed solution relies on.

Model Differences Metamodel and model versioning is achieved through a model-based representation to store the differences between an old and a new (meta)model revision. Such representation is based on an enriched version of an existing work [CDP07] which introduced a technique for the storage of model evolution relying on the partitioning of the manipulations into three basic operations: *additions*, *deletions*, and *updates* of model elements. Since a metamodel is a model itself, it is possible to adopt the same mechanism also for the representation of metamodel evolution, as already clarified in [CDP09]; therefore, we will generically write about metamodels and models taking into account that in the case of metamodel evolutions the metamodel plays the role of the model and its metamodel is the metametamodel.

A differences language is obtained through the automated extension of the metamodel the models conform to: for instance, developers working with metamodel $\texttt{MM}_a$ will modify models and produce new instances conforming to it, as described in Section 4. Therefore, the language able to represent model manipulations will be generated by means of an appropriate extension of $\texttt{MM}_a$, called $\texttt{DiffMM}_a$. More specifically, each non-abstract metaclass `MC` in the metamodel induces three specialisations, namely `AddedMC`, `DeletedMC`, and `ChangedMC`, which are exploited to represent the additions, deletions, and changes, respectively, of such metaclass. Moreover, an `updatedElement` association connects an updated metaelement with its corresponding new version[3].

In order to be able to apply differences in the sense of added and deleted elements, a reference to their containing element must be maintained; therefore a reference, namely `containedBy`, is introduced to point to the container of the corresponding `AddedMC` or `DeletedMC`. In the same way, $\texttt{MM}_b$ metamodel is extended to obtain the corresponding $\texttt{DiffMM}_b$ difference language; an appropriate extension of the Ecore metamodel is generated to represent revisions of metamodel, e.g., to store differences between $\texttt{MM}_a$ and $\texttt{MM}_b$.

As discussed in [CDP07], the differences representation approach enjoys a number of interesting properties that allow the appropriate versioning of models and the retrieval of previous revisions; nonetheless, as an immediate consequence of dealing with a mixture of models conforming to different metamodels a problem arises because of the incomparability between concurrent revisions. In particular, by recalling the scenarios described in Section 4, manipulations made on the repository and the ones locally operated may pertain to different languages, i.e., $\texttt{DiffMM}_a$ and $\texttt{DiffMM}_b$, respectively. In this respect, we propose a general solution based on a difference metamodels merging technique able to produce a *merged difference metamodel* including concepts from both the versions. It is worth mentioning that this comparability problem is quite common when facing metamodel evolution and trying to represent the modifications of the migrated models. In this sense, relaxing language constraints to build-up a sort of *lingua franca* is an accepted practice [HBJ08, WKS+10]. It is also

[3]For further details on the representation approach and the motivations underlying some design choices the reader is referred to [CDP07].

important to notice that in general current metamodelling environments do not allow to use models that do not strictly conform to corresponding metamodels. Therefore, even if metamodel agnostic approaches for differences detection and representation may perform better at the beginning, in the long run they could entail several inefficiencies and accuracy issues (please refer to Section 7 for a detailed discussion of this problem).

The adopted differences representation technique discloses the possibility to exploit automated (and precise) co-evolution mechanisms, as detailed in the next paragraph.

Metamodels Evolution and Models Co-Evolution As stated above, metamodels are models themselves, therefore the reasoning made for model versioning techniques is completely applicable in this case too. However, for differencing purposes it could be worth to declaratively state the evolution metamodels were subjected to (i.e., by avoiding automated detection mechanisms), since the order of magnitude of metamodels is in general much smaller than their model instances, and an erroneous metamodel evolution detection can cause unpredictable side-effects on the corresponding model migrations [CDP09]. Once the metamodel evolution is properly stored through the differences representation approach, (semi-)automated mechanisms are enabled to migrate instances conforming to the previous version of the metamodel toward the newer version. In particular, the method introduced in [CDP09] takes as input the differences between two metamodel versions and generates a corresponding co-evolution transformation migrating the existing model instances. Alternatively, the approach proposed in [DIP11] declaratively prescribes the co-evolution operations to be performed as corresponding to metamodel manipulations described by a textual form of the adopted metamodel independent differences representation.

At this point, it is important to notice that, in our approach, based on the mechanisms introduced in [CDP09], model migrations due to metamodel evolution and model manipulations cannot conflict with each other, since the two evolutions happen sequentially and the latest (in terms of application time) always overwrites the previous. Therefore, if modifications are compatible across different revisions of the metamodel, they are committed accordingly, even after migration steps. If manipulations are not compatible, e.g., because an entity has been removed, the changes will stay valid in the local working copy. Subsequently, whenever a migration step would be operated those modifications would lose their meaning and therefore ignored; a further discussion concerning these issues is given in Section 7.

In the following, we first describe the mechanism to merge difference metamodels and hence allow a common representation of ongoing changes by the different languages; then, we explain how such a method can help in solving the issues arising in the scenarios presented in Section 4.

Merging Metamodels Migrating models across different metamodel versions is made possible by the adoption of a *lingua franca*, as aforementioned. In our case, such bridging language derives from a merging operation on the involved difference metamodels (i.e., previously introduced $\texttt{DiffMM}_a$ and $\texttt{DiffMM}_b$) that results into a new merged difference metamodel `DiffMM` which is capable to represent all the information conforming to both $\texttt{DiffMM}_a$ and $\texttt{DiffMM}_b$.

The algorithm for creating `DiffMM` resembles the one proposed in [WKS$^+$10] and consists of two main steps:

1. *Identification and merge of mergeable metaclasses:* in this first step, the classes

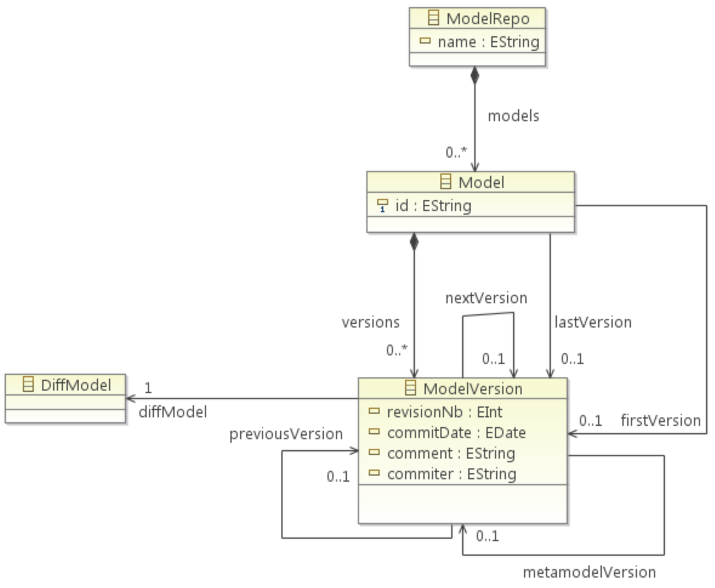


Figure 5 – Repository metamodel

of $\texttt{DiffMM}_a$ and $\texttt{DiffMM}_b$ are compared and, if they have same name, they will be merged into one single class in `DiffMM`. The new class will get the union of the features (i.e., attributes and references) and superclasses of both $\texttt{DiffMM}_a$ and $\texttt{DiffMM}_b$; in fact, while for classes the merge condition is sameness in their name, for merging features a total sameness in terms of, for instance, *multiplicity*, *type*, *unique* constraint is needed;

2. *Inclusion of the non-mergeable metaclasses from $\texttt{DiffMM}_a$ and $\texttt{DiffMM}_b$:* the metaclasses in $\texttt{DiffMM}_a$ that do no have a correspondent in $\texttt{DiffMM}_b$ and vice versa are directly included to `DiffMM` with their own features.

The merging operation is carried out by a model-to-model transformation defined in Operational QVT[4] which takes as input the two difference metamodels and generates the merged difference metamodel. The interested reader can refer to Appendix A where the QVTo transformation is depicted.

Model Versions In order to describe how we represent the different model versions three assumptions have to be clarified: (i) all model versions are stored in a shared model repository, (ii) a model is identified without analysing its content but rather using, for instance, a global unique identifier and, similarly, (iii) a model version is identified by a revision number that we automatically assign to it together with the model unique identifier. In Figure 5 the repository metamodel used to represent model versions is depicted. While branching and merging of concurrent versions is a useful feature of a model repository, our main objective is to manage concurrent model and metamodel evolution thus focusing on merging and leaving branching support as desirable future enhancement of the current solution (as discussed in Section 7).

[4]`http://wiki.eclipse.org/M2M/Operational_QVT_Language_(QVTO)`

Besides revision number, standard metadata such as committer name and commit date are associated to model revisions in order to allow their ordering in the repository. As described in the previous section, we use a model-based differences representation to individuate and represent the performed changes between two model versions. As a result, each model revision is associated to its corresponding difference model which describes differences between the current and previous versions.

Moreover our approach introduces the novel solution of tracking the metamodel version related to each model version in order to correctly achieve concurrent versioning; this means that no assumptions are made about the metamodel which is, in fact, versioned as any other model. Thanks to such solution, the model repository is able to manage a non fixed, though finite, number of metamodelling levels while still being able to track conformance between models and metamodels.

Additionally, the metamodel placed at the root of the metamodelling hierarchy is defined as conforming to itself in order to avoid infinite hierarchy levels. The repository itself is in turn a model which conforms to the presented repository metamodel and such characteristic allows to take advantage of standard model databases such as *Teneo*[5] to implement the repository.

5.3 General Approach

Our approach aims at managing concurrent versioning of metamodels and models without jeopardising abstract and concrete syntax of models as well as local saving formats. The whole process is meant to be transparent to the model management tools. While providing solutions for addressing the three scenarios presented in Section 4, attention is also paid into possible optimisations of the repository size. When evolving metamodels, part of the old information might disappear in the newer version; even in such cases, the versioning approach aims at preserving such information in order to be able to move backwards to older metamodel versions when requested. To this end, our approach is based on computing (meta)model differences between (meta)model versions and tracking compatibility links between model and metamodel versions. Such differences are stored in the repository in terms of difference models as well as the difference metamodels they conform to. Regarding the notation that we will use to distinguish these artefacts: $\texttt{DiffM}_{xy}$ will represent the differences (model) between models $\texttt{M}_x$ and $\texttt{M}_y$, $\texttt{EvoMM}_{xy}$ the differences between metamodels $\texttt{MM}_x$ and $\texttt{MM}_y$, `DiffMM` the merged difference metamodel to which $\texttt{DiffM}_{xy}$ conforms to, while `DiffMMM` the difference metametamodel to which $\texttt{EvoMM}_{xy}$ conforms to. Each (meta)model conforms to the root language `MMM` (e.g., Ecore, MOF, EMOF, etc.). The ability of managing traceability links is a prerequisite to be able to merge concurrent modifications made on different metamodel versions. Our versioning approach aims at supporting the standard versioning actions (i.e., Commit, Update, Revert and Checkout) on both models and metamodels. The *commit* process depicted in Figure 6 where the $\texttt{M}_e$ has evolved into $\texttt{M}_f$ (and $\texttt{MM}_e$ into $\texttt{MM}_f$) in the workspace and then such evolution is stored in the repository, is composed by the following steps:

1. First, metamodels (e.g., $\texttt{DiffMM}_e$ and $\texttt{DiffMM}_f$) in charge of representing model differences are derived from the original metamodels (e.g., $\texttt{MM}_e$, $\texttt{MM}_f$). Since there has been a metamodel evolution (either user has modified the metamodel or a new metamodel is available in the repository), two difference metamodels have been generated. If the destination metamodel version is not present in the

[5] `http://wiki.eclipse.org/Teneo`

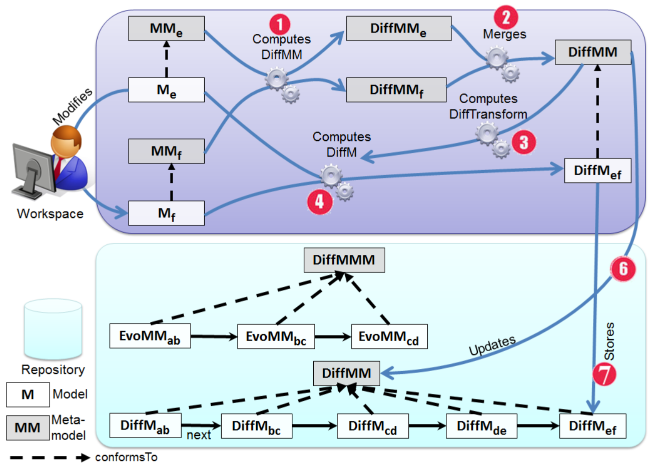


Figure 6 – Overview of the commit process

workspace, it is retrieved from the repository. We assume that the base[6] model version and its corresponding metamodel version are in the user workspace.

2. The generated difference metamodels are merged in a single difference metamodel (DiffMM) able to represent differences in conformance with both $\texttt{DiffMM}_e$ and $\texttt{DiffMM}_f$;

3. A model transformation (Computes DiffM) which computes model differences is generated from the merged difference metamodel (DiffMM);

4. This differences computation is performed on the two model versions ($\texttt{M}_e$, $\texttt{M}_f$). It produces a difference model ($\texttt{DiffM}_{ef}$) which conforms to the merged difference metamodel (DiffMM);

5. In case of *metamodel evolution*, the commit process is applied to the metamodels (base and new versions) in the same way as for the models. More specifically an evolution model $\texttt{EvoMM}_{ef}$ is generated in the workspace in the same way as for $\texttt{DiffM}_{ef}$ and stored in the repository;

6. At this point we use the information brought by DiffMM to upgrade the difference metamodel used to represent model differences (DiffMM) in the repository;

7. Now we can store the difference model $\texttt{DiffM}_{ef}$ in the repository as well as the related model version metadata and the traceability link to the corresponding metamodel version.

A novelty brought by our approach is the derivation of the differences computation algorithm from the involved base and new metamodels (as discussed in Section 6). The *update* process is shown in Figure 7. It moves from a source model version in the workspace (i.e., $\texttt{M}_c$), to a destination model version available in the repository (given for instance by the difference models $\texttt{DiffM}_{cd}$ and $\texttt{DiffM}_{de}$). Moreover, possible

[6]For *base* (meta)model is meant the original version while with *new* (meta)model is meant an evolution of the base version.

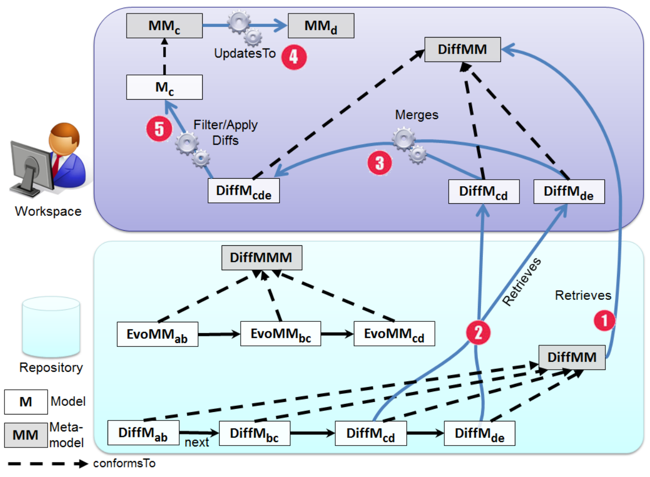


Figure 7 – Overview of the update process

update of the workspace metamodel (from $\mathtt{MM}_c$ to $\mathtt{MM}_d$) is included in flow. The update process is composed by the following steps:

1. First, the metamodel (`DiffMM`) in charge of representing model differences is retrieved from the repository;
2. The difference models ($\mathtt{DiffM}_{cd}$, $\mathtt{DiffM}_{de}$) representing modifications applied somewhere else to $\mathtt{M}_c$ are retrieved into the workspace;
3. Such difference models are merged into one difference model ($\mathtt{DiffM}_{cde}$);
4. If the metamodel needs also to be updated, the update process is applied to it in the same way as for the models but using, e.g., $\mathtt{EvoMM}_{cd}$ for evolving from the current workspace version $\mathtt{MM}_c$ to the latest repository version $\mathtt{MM}_d$);
5. At this point the merged difference model is filtered to only account the modifications relevant for the metamodel version present in the workspace which are then applied to the workspace model version ($\mathtt{M}_c$).

The *revert* action is a specific update case where the destination version is the base version. The *checkout* action is also a specific case of the update process where source version represents an empty model; therefore we assume that the first model version is represented by an empty model.

Assuming that exchange of updates only happens via repository, which contains the complete history of difference metamodels and models, allows us to minimise the loss of information especially when propagating changes among workspaces working on models conforming to different language versions.

Differences Filtering and Application Both in the commit and the update processes the application of differences, in particular when dealing with different metamodel versions, has to be filtered in order to consider only those differences that, conceiving a given metamodel version, can be applicable to the models conforming to it. In fact, by merging difference (meta)models, the whole set of represented differences is available yet not fully applicable to any (meta)model version. Let us consider again the artefacts depicted in Figure 1, the difference metamodels $\texttt{DiffMM}_a$ and $\texttt{DiffMM}_b$ and the merged difference metamodel `DiffMM` which is capable to represent the union of the information conforming to $\texttt{DiffMM}_a$ and $\texttt{DiffMM}_b$. Let us suppose we want to commit ${\texttt{M}_a}^2$ to the repository, differences between ${\texttt{M}_a}^1$ and ${\texttt{M}_a}^2$ are represented in a model $\texttt{DiffM}_x$ which conforms to `DiffMM`. Some of the differences represented in $\texttt{DiffM}_x$ are applicable to $\texttt{M}_b$ since they involve metaelements of $\texttt{DiffMM}_b$ (and therefore $\texttt{MM}_b$); the filtering operation detects such applicable differences and ignores the ones regarding metaelements present in $\texttt{DiffMM}_a$ but not in $\texttt{DiffMM}_b$. The filtering mechanism is achieved by construction since the differences application transformation is automatically generated ad-hoc for the target metamodel (e.g., $\texttt{DiffMM}_b$) as described in Section 6.4. That is to say that the generated transformation will be only composed by rules applicable to those metaelements contained in $\texttt{DiffM}_b$ and thereby be able to apply only the differences concerning them. While the filtering is achieved by construction when generating the differences application transformation, the actual changes application is achieved by navigating the difference model $\texttt{DiffM}_x$ and apply modifications represented in terms of `AddedMC`, `DeletedMC`, and `ChangedMC` to $\texttt{M}_b$ by:

- Adding `AddedMC` elements: new model elements introduced in $\texttt{M}_{a^2}$ are created in $\texttt{M}_b$ with particular attention to containments for correct placing of the new elements;
- Deleting `DeletedMC` elements: models deleted in ${\texttt{M}_a}^2$ are removed (if still present) from $\texttt{M}_b$ with particular attention to containments for correct deletion of the elements and eventual propagation of the deletion procedure to contained elements;
- Modifying `ChangedMC` elements: modifications applied to elements ${\texttt{M}_a}^2$ are applied to $\texttt{M}_b$, if the involved elements are still in place.

6 Implementing the Concurrent Versioning

In this section a detailed description of the implementation for the proposed solution is given and applied to the *School* language example. The considered scenario is the update of a workspace model from an evolved repository model conforming to different versions of the language (Figure 2). In this example we will consider one workspace and a repository since the number of workspaces does not affect the proposed process; in fact, evolutions are not propagated directly among workspaces but always via the common repository. The solution has been implemented on the Eclipse Modeling Framework (EMF) and a prototype is available as a set of plugins[7].

6.1 A Running Example: the School Metamodel

In order to describe and validate the proposed approach we will take advantage of a running example defined using EMF: the *School* metamodel. In Figure 8 we depict the workspace version of the *School* metamodel, referred to as *School workspace version*, and an evolved version of it which is the current version in the repository, namely

[7]For the interested reader the implemented prototype plugins are available for download at `http://www.mrtc.mdh.se/~acicchetti/concurrentVersioning.php`

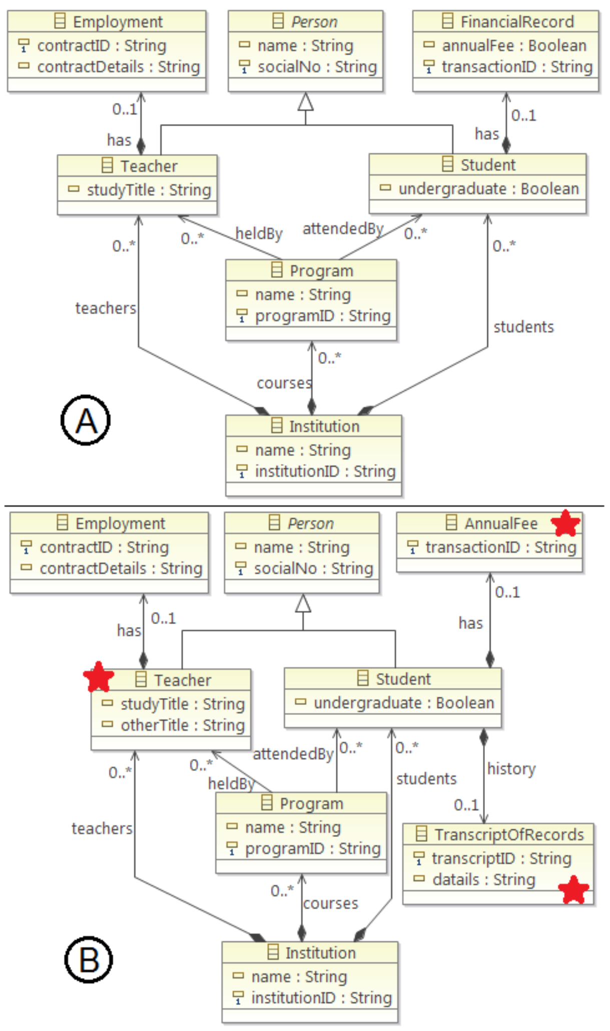


Figure 8 – School workspace version (A) and repository version (B)

School repository version (evolved metaelements are marked with a star in the latter). The *School workspace version* language allows to model a simple school system in which we have the following main elements:

- `Institution`: it represents the school and contains elements of type `Program`, `Teacher` and `Student`. `InstitutionID` is considered the unique identifier;
- `Program`: it represents a program offered by the school and has references to both teachers and students involved in it. `ProgramID` is the attribute acting as unique identifier;
- `Teacher`: it represents a teacher employed by the school and may contain an element of type `Employment` representing his contract. `Teacher` specialises the abstract metaclass `Person`, inheriting the attribute `socialNo` which acts as unique identifier;

- `Student`: it represents a student enrolled in the school and may contain an element of type `FinancialRecord` representing the annual fee to be paid to be enrolled. As well as `Teacher`, `Student` also specialises the abstract metaclass `Person`, inheriting the attribute `socialNo` which again acts as unique identifier.

The modifications making the language to evolve in the repository version are (i) the introduction of a new element `TranscriptOfRecords` contained by `Student`, (ii) the deletion of `FinancialRecord` replaced by `AnnualFee`, and (iii) the addition of a new attribute, namely `otherTitle`, in `Teacher`. In the remainder of the article we rely on these two different versions of the *School* language for describing our versioning approach.

6.2 Model Repository

Our concurrent versioning system is built on top of the MORSA [EPSCGM11] EMF model repository; it stores models into a Mongo database which does not rely on relational schemas. This characteristic increases flexibility in managing metamodel evolutions. The choice of EMF, which defines the Ecore language by means of itself, allows us to satisfy the assumption of a finite metamodelling hierarchy. As previously mentioned, our approach only stores the model version differences. The repository content is an EMF model which conforms to the repository metamodel depicted in Figure 5. While model elements are singularly loaded on demand during the navigation in order to only load selected model version metadata, difference models content is always loaded as a whole. Since metamodels are in turn models, the nature (being a metamodel) of the model is transparent to the versioning mechanism and such characteristic enables the ability by our system to manage versioning for a non fixed, though finite, number of metamodeling levels. Transparency of the versioning process to other modelling tools is achieved by storing the version metadata in a separated file to which the model file does not have any reference. In order to improve performance, a copy of the base model version is also stored in a local hidden directory in a similar way as done by Subversion. During a model version update, if the destination model version relies on a different metamodel revision than the current, the user is asked to choose if the metamodel version must be updated.

6.3 Model Differences Detection

The computation of the differences between model versions is based on transformations specified in terms of the Epsilon comparison, merge and transformation languages[8] in a similar way as done in [CCL11]. A set of rules defined using the Epsilon Comparison Language (ECL) is applied to two different revisions of the model. These rules are in charge of identifying matching elements between the two models by comparing their elements through their unique identifier; once a match is found, possible changes among the revisions are also caught and stored. Using identifiers allows to correctly find the two corresponding elements between initial and modified version of the model and explore their properties to detect possible changes. Once the matching phase has been completed, a set of rules defined using the Epsilon Merge Language (EML) are in charge of transparently taking the ECL transformation results and thereby generate the difference model in terms of changed/original element pairs. If the ECL result structure contains information about a modification which affected the

[8]`http://eclipse.org/gmt/epsilon/`

element, the changed element is created and linked to the original within the rule's body by using EOL expressions. As well as modified elements, deleted and added elements have to be identified too in order to produce a complete difference model; this step is performed through specific rules defined using the Epsilon Transformation Language (ETL). The results of the differences computation operations are finally stored in a difference model, conforming to the merged difference metamodel automatically generated as described in Section 5.2, ready to be used for the differences application to the model version to be updated. In Figure 9 we depict an excerpt of the merged difference metamodel representing the possible changes applicable to both workspace and repository revision on the elements contained by `Student`. In Figure 10 original

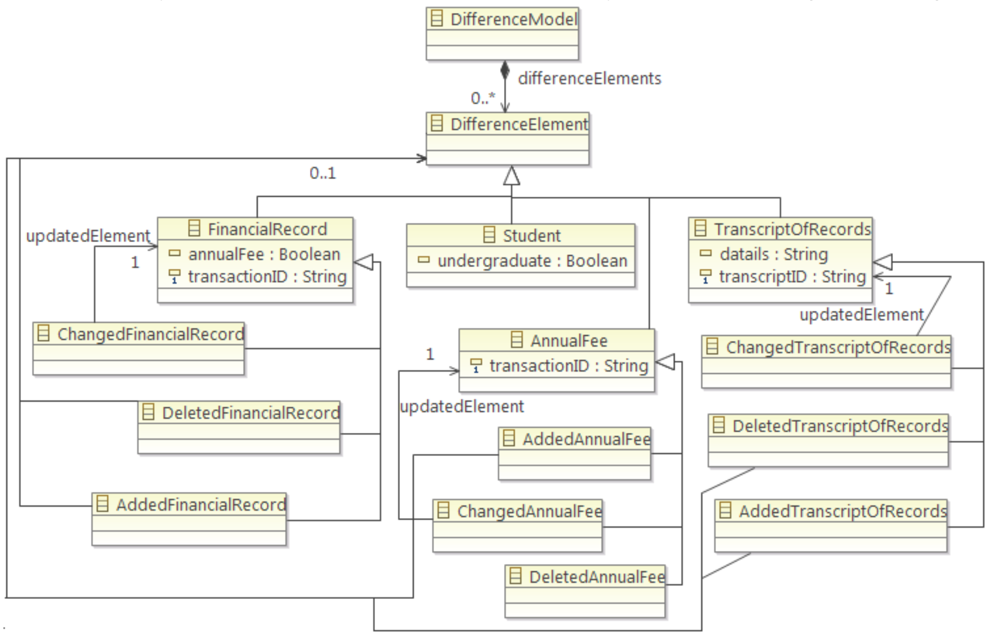


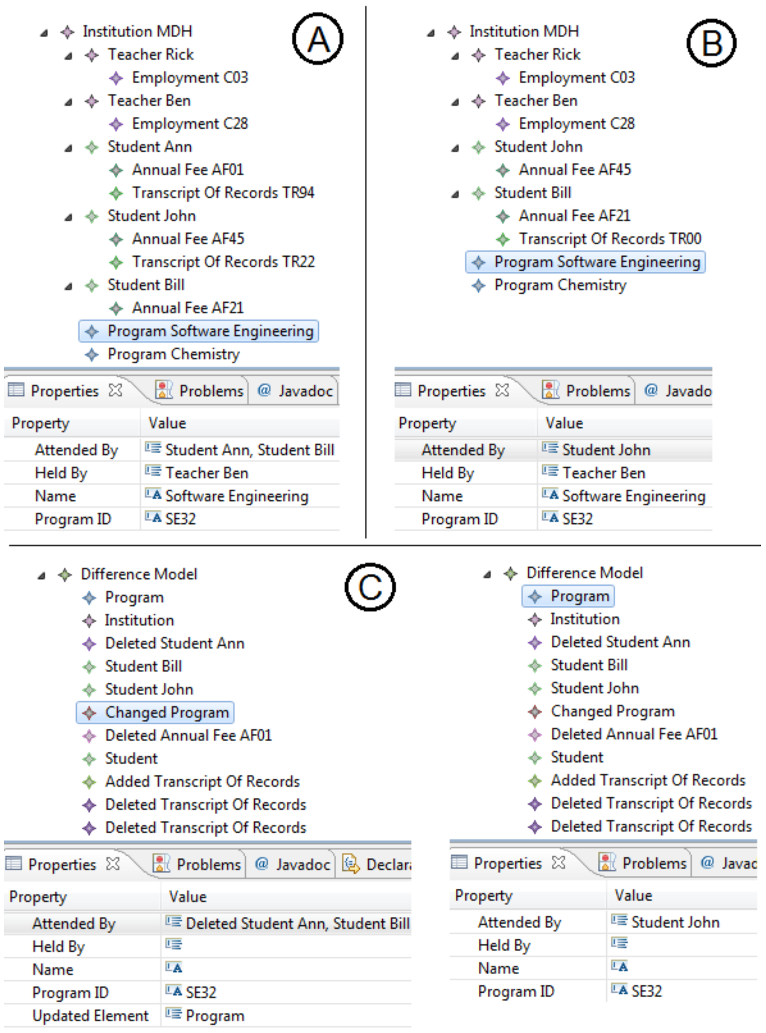
Figure 9 – Excerpt of the merged difference metamodel for the School languages

(Figure 10A) and modified (Figure 10B) versions of the *School repository* model are depicted together with the computed difference model (Figure 10C). The operated changes, properly represented in the difference model, are:

1. element *Student Ann* is deleted;
2. element *AnnualFee AF01* is deleted;
3. element *TranscriptOfRecords TR94* is deleted;
4. element *TranscriptOfRecords TR22* is deleted;
5. element *TranscriptOfRecords TR00* is added;
6. element *Program Software Engineering* is modified through the its reference `attendedBy` that from pointing to *Student Ann*, *Student Bill* in the original repository version, points to *Student John* in the modified repository version.

The Epsilon transformations for differences calculation and difference model creation are generated by a higher-order transformation (HOT), defined in the Acceleo model-to-text transformation language[9], which creates the ad-hoc differences calculation transformations starting from any metamodel defined in Ecore and in which each metaclass has at least one specified metaattribute acting as unique identifier. In

[9]`http://www.acceleo.org`



Institution MDH
- Teacher Rick
  - Employment C03
- Teacher Ben
  - Employment C28
- Student Ann
  - Annual Fee AF01
  - Transcript Of Records TR94
- Student John
  - Annual Fee AF45
  - Transcript Of Records TR22
- Student Bill
  - Annual Fee AF21
- Program Software Engineering
- Program Chemistry

Property	Value
Attended By	Student Ann, Student Bill
Held By	Teacher Ben
Name	Software Engineering
Program ID	SE32

Institution MDH
- Teacher Rick
  - Employment C03
- Teacher Ben
  - Employment C28
- Student John
  - Annual Fee AF45
- Student Bill
  - Annual Fee AF21
  - Transcript Of Records TR00
- Program Software Engineering
- Program Chemistry

Property	Value
Attended By	Student John
Held By	Teacher Ben
Name	Software Engineering
Program ID	SE32

Difference Model
- Program
- Institution
- Deleted Student Ann
- Student Bill
- Student John
- Changed Program
- Deleted Annual Fee AF01
- Student
- Added Transcript Of Records
- Deleted Transcript Of Records
- Deleted Transcript Of Records

Property	Value
Attended By	Deleted Student Ann, Student Bill
Held By	
Name	
Program ID	SE32
Updated Element	Program

Difference Model
- Program
- Institution
- Deleted Student Ann
- Student Bill
- Student John
- Changed Program
- Deleted Annual Fee AF01
- Student
- Added Transcript Of Records
- Deleted Transcript Of Records
- Deleted Transcript Of Records

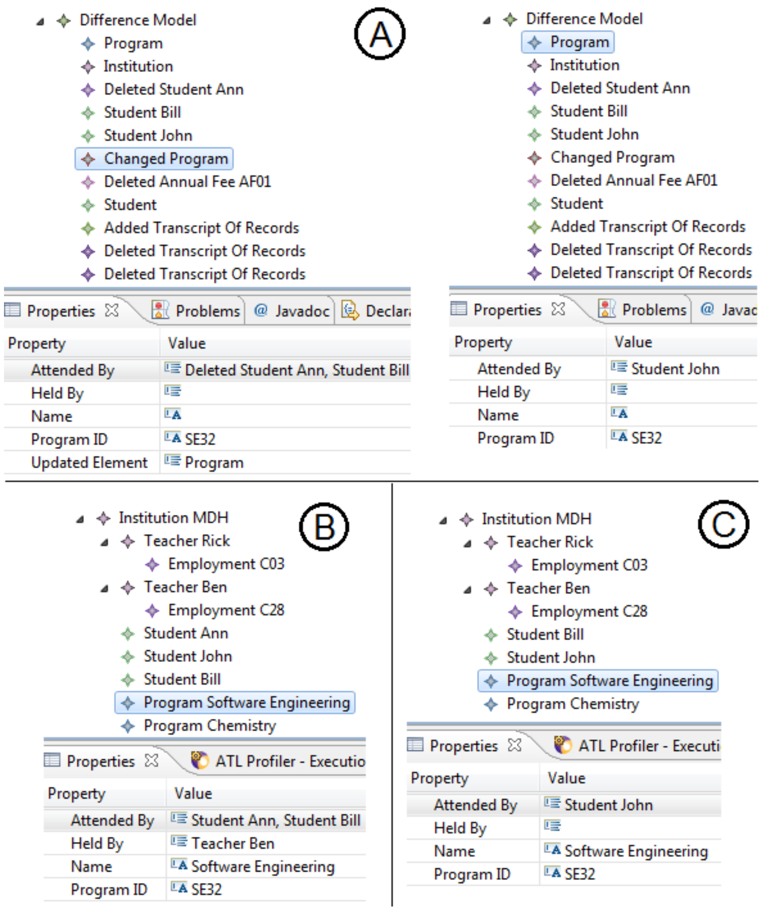
Property	Value
Attended By	Student John
Held By	
Name	
Program ID	SE32

Figure 10 – School repository versions (A-B) and difference model (C)

Appendix B an excerpt of the HOT is given (left-hand side) as well as the EML rules for detecting changed elements to that result from its execution (right-hand side).

6.4 Model Differences Application

Once changes operated in the repository have been properly detected and represented, they have to be applied to the current model revision in the local workspace conforming to a different metamodel version. Since the difference model conforms both to the local difference metamodel and to the merged difference metamodel, the changes it represents can be applied to any of the model revisions. In fact model changes that would not have sense in the local metamodel version are not considered in the differences application transformation. This is possible by a HOT, defined by means of Acceleo, that generates the QVTo differences application transformation starting from the merged difference metamodel and the local (target) metamodel. In this way,



- Difference Model
  - Program
  - Institution
  - Deleted Student Ann
  - Student Bill
  - Student John
  - Changed Program
  - Deleted Annual Fee AF01
  - Student
  - Added Transcript Of Records
  - Deleted Transcript Of Records
  - Deleted Transcript Of Records

Property	Value
Attended By	Deleted Student Ann, Student Bill
Held By	
Name	
Program ID	SE32
Updated Element	Program

- Difference Model
  - Program
  - Institution
  - Deleted Student Ann
  - Student Bill
  - Student John
  - Changed Program
  - Deleted Annual Fee AF01
  - Student
  - Added Transcript Of Records
  - Deleted Transcript Of Records
  - Deleted Transcript Of Records

Property	Value
Attended By	Student John
Held By	
Name	
Program ID	SE32

- Institution MDH
  - Teacher Rick
    - Employment C03
  - Teacher Ben
    - Employment C28
  - Student Ann
  - Student John
  - Student Bill
  - Program Software Engineering
  - Program Chemistry

Property	Value
Attended By	Student Ann, Student Bill
Held By	Teacher Ben
Name	Software Engineering
Program ID	SE32

- Institution MDH
  - Teacher Rick
    - Employment C03
  - Teacher Ben
    - Employment C28
  - Student Bill
  - Student John
  - Program Software Engineering
  - Program Chemistry

Property	Value
Attended By	Student John
Held By	
Name	Software Engineering
Program ID	SE32

Figure 11 – Difference model (A) and School workspace model before (B) and after (C) the differences application

the resulting transformation will only consider differences regarding metaelements proper of the *School Local* metamodel given as input to the HOT; the filtering mechanism is therefore automatically achieved when generating the differences application transformation.

The generated QVTo transformation is able to apply the changes described in Section 6.3 and represented in the *Repository* difference model, given in input, to the *School workspace* model by means of in-place modifications. More specifically, the applied changes are (1) the deletion of element *Student Ann* and (6) modification of element *Program Software Engineering* while changes (2), (3), (4) and (5) are not applied since `AnnualFee` and `TranscriptOfRecords` are not part of the workspace metamodel version. Difference models and *School workspace* model before and after the differences application are depicted depicted in Figure 11. In Appendix C an excerpt of the HOT is given (left-hand side) as well as the QVTo transformation for differences application that results from its execution (right-hand side) concerning the metaclass *Program* in the School metamodel.

7 Discussion

This section discusses relevant issues related to the proposed solution and the lessons learnt by applying it to a set of case-studies.

In file-based versioning, the *branching and merging* technique allows to isolate changes into separate development lines, called branches, within the same workspace. Changing files on a branch does not propagate those changes to other branches. Instead, changes can be moved on demand from one branch to another by merging operations. In our approach we propose a similar process if considering the different workspaces as single branched; in fact the developers keep on working on their own branch (with respect to the metamodel version) and when they decide to either commit or update, merging operations allow the partial propagation of changes. Moreover, branches are unified whenever metamodel revisions are re-aligned to the latest version available in the shared repository, i.e., when the metamodel is updated in the local working copy and the existing models are migrated. An interesting future step could be to add an additional local versioning by enabling branching and merging within each single workspace and then use the same process as the one presented for versioning across different workspaces. In this respect a fundamental difference with respect to the traditional branching and merging techniques is that, in our approach, model migrations due to metamodel evolution cannot conflict with model manipulations since they happen sequentially. Therefore, either the model is first migrated and then modifications are applied to it or vice versa. In any case, due to metamodel evolution loss of information might happen. For instance, if a certain metaclass has been removed, any operation performed on its instances will be lost after the migration operation. A possible solution to this issue could be to exploit model patching methods available with the adopted difference representation technique [CDP10] for attempting the propagation of lost changes, whenever meaningful.

Additional considerations to be done on the migration step are related to metamodel evolutions for which user intervention is unavoidable. They are defined as *unresolvable changes* [SK04] and make automated model co-evolution not feasible: e.g., the addition of a mandatory metaattribute can require not only its creation in all the existing model instances, but also the choice of a corresponding value for it [CDP09]. As a consequence, manual migration decisions could cause inconsistencies between local copies and the repository version, since different users could make different choices. Nonetheless, those inconsistencies would be discovered at the first commit or update attempt and required to be fixed.

The adoption of the differences representation mechanism in [CDP07] discloses the opportunity to exploit a set of existing utilities, notably automated co-evolution and model patching. It is important to notice that there exist additional techniques based on a metamodel tailored representation of differences allowing for a more general co-evolution support (though not automated), including transformations and editing tools [DIP11]. Nonetheless, an alternative approach could have been the adoption of a metamodel agnostic differences detection and representation, as adopted in model comparison (e.g., [LGJ07, Tou]) and model differences representation (e.g., [RV08]). Despite such approaches would have avoided the issue of merging metamodels, (meta)model versioning would have resulted less efficient and accurate. In fact, current techniques typically adopt "tagging" mechanisms of model elements (notably added, updated, deleted) that do not make differences context-independent. In other words, the approaches need to carry extra-information about the involved artefacts to allow an appropriate interpretation of the evolution [CDP07]. Such need hinders, for instance, the exploitation of patching techniques, since differences are difficult to abstract from the detection context. Moreover, comparison methods have to rely on structural similarity measures that tend to be less efficient and accurate when addressing differences among different metamodels [KDPP09]. Finally, the problem of which metamodel to adopt to represent the current model version should be solved, since current metamodelling frameworks, like EMF, do not allow to work with models which are not strictly conforming to a corresponding metamodel.

The proposed approach has been validated against small to medium sized (up to 50 elements for metamodels and 100 for models) (meta)models built using different modelling languages expressed using EMF. The complexity of the process requires an industrial sized validation for verifying possible scalability issues that did not arise in our experiments. In order to allow evaluation of a possible applicability of the approach to the reader's specific case-study, we provide the limit behaviour of each involved transformation in relation to the input (meta)model elements (n=number of elements of the biggest model, m=number of metaelements of the biggest metamodel) in Table 1.

Transformation	**Language**	**Complexity**
Metamodels Merging	QVTo	$O(n^2)$
HOT Differences Calculation	Acceleo	$O(n^2)$
HOT Differences Application	Acceleo	$O(n^2)$
Generation Difference Metamodel	ATL	$O(n)$
Differences Calculation	QVTo	$O(m^2)$
Difference Application	QVTo	$O(m)$

Table 1 – Limit behaviour of the involved transformations

8 Conclusions and Future Work

In this article we addressed the problem of concurrent versioning of models and metamodels. These tasks are often seen independent of each other and hence addressed separately; our claim is that they affect and eventually hinder each other thus demanding a concurrent management. Through a scenario-driven description, the issues arising when attacking the problem of concurrent versioning of models and metamodels are shown; then, for each specific scenario, a solution is proposed. The implementation of the versioning system is described step-by-step exploiting the scenario 1 and taking advantage of the *School* language example, although the validation of the solution has been performed on several different languages.

Our main contribution is a solution for managing misaligned evolution of models and their respective metamodels. The proposed approach allows the user to decide when to switch to the most recent metamodel version; that is to say that in any case, even before upgrading to the most recent version, the user can make use of the model versioning system.

The common denominator of the proposed solutions is the exploitation of model comparison mechanisms for calculating changes between different model and metamodel versions as well as applying such changes to local or remote versions to be committed or updated. The proposed solution takes advantage of the concept of a *merged difference metamodel* that enables the contemporary representation of manipulations coming from both different language versions. Moreover, thanks to the differences representation mechanisms and the generation of differences calculation and application transformations by means of generic (in the sense that any Ecore metamodel can be fed as input) higher-order transformations, the versioning system is able to deal with any language defined in Ecore as well as with a non-fixed, yet finite, number of metamodelling levels.

The proposed solution poses a number of technical difficulties, mainly due to the intrinsic characteristics of co-evolution problems. In fact, in general some migration cases cannot be fully automated and require user intervention. Additionally, in order to ensure the consistency between modifications and the correctness of commit/update operations, migration transformations have to be reproduced each time it is needed. In other words, migrations between different revisions of a metamodel have always to perform the same co-evolution operations. Therefore, whenever user's input would be needed in the migration phase, it should be stored and replicated when the co-evolution is re-applied. Moreover, the downgrading transformation should be the exact inverse mapping of the upgrading one, thus posing additional problems to the consistency and correctness of the process. Further enhancements of the mechanisms adopted for evolution and migration are already ongoing. Moreover, the proposed approach will be applied (adapted and extended if needed) to other metamodel-dependent artefacts such as transformations and graphical editors.

The approach has been validated against examples built by means of different modelling languages coming from our experience in industrial projects [ART09, MRT06]. Nonetheless, the complexity of the process requires a thorough validation against industrial sized scenarios; in particular, we are interested in verifying whether scalability issues would arise when dealing with large development groups and a particularly high number of model elements.

A QVTo transformation for Merging Difference Metamodels

```
modeltype Ecore uses "http://www.eclipse.org/emf/2002/Ecore";
transformation merger(in MM1 : Ecore, in MM2 : Ecore, out MM_Merged : Ecore);
main() {
    MM1.objectsOfType(Ecore::EPackage)->asOrderedSet()->first().map packageToPackage();
    MM2.objectsOfType(Ecore::EClass)->forEach(c){c.checkClass();    };
    MM2.objectsOfType(Ecore::EDataType)->forEach(d){d.checkDataType();  };
    MM_Merged.objectsOfType(Ecore::EClass)->forEach(c){completeClass(c);    };
}
helper completeClass(inout c : Ecore::EClass){
    //add supertypes from both input MMs
    MM1.objectsOfType(Ecore::EClass)->select(name = c.name)->asOrderedSet()->first()[Ecore::EClass].eSuperTypes->forEach(st){
        c.eSuperTypes+=MM_Merged.objectsOfType(Ecore::EClass)->select(name = st.name)->asOrderedSet()->first(); };
    MM2.objectsOfType(Ecore::EClass)->select(name = c.name)->asOrderedSet()->first()[Ecore::EClass].eSuperTypes->forEach(st){
        if(c.eSuperTypes->select(name = st.name)->size() = 0)then{
            c.eSuperTypes+=MM_Merged.objectsOfType(Ecore::EClass)->select(name = st.name)->asOrderedSet()->first(); }endif;};
    //add ereferences and eattributes from both input MMs
    MM1.objectsOfType(Ecore::EClass)->select(name = c.name)->asOrderedSet()->first()[Ecore::EClass].eAttributes->forEach(esf){
        if(c.eAttributes->select(name = esf.name)->size() = 0)then{
            c.eStructuralFeatures+=esf.map attrToAttr();    }endif;};
    MM1.objectsOfType(Ecore::EClass)->select(name = c.name)->asOrderedSet()->first()[Ecore::EClass].eReferences->forEach(esf){
        if(c.eStructuralFeatures->select(name = esf.name)->size() = 0)then{
            c.eStructuralFeatures+=esf.map refToRef();  }endif;};
    MM2.objectsOfType(Ecore::EClass)->select(name = c.name)->asOrderedSet()->first()[Ecore::EClass].eAttributes->forEach(esf){
        if(c.eAttributes->select(name = esf.name)->size() = 0)then{
            c.eStructuralFeatures+=esf.map attrToAttr();    }endif;};
    MM2.objectsOfType(Ecore::EClass)->select(name = c.name)->asOrderedSet()->first()[Ecore::EClass].eReferences->forEach(esf){
        if(c.eStructuralFeatures->select(name = esf.name)->size() = 0)then{
            c.eStructuralFeatures+=esf.map refToRef();  }endif;};
}
mapping Ecore::EPackage::packageToPackage(): Ecore::EPackage{
    name:=self.name;
    nsURI:=self.nsURI;
    nsPrefix:=self.nsPrefix;
    //add elements from MM1
    eClassifiers+=self.allSubobjectsOfType(Ecore::EClass)[Ecore::EClass]->map classToClass();
    eClassifiers+=self.allSubobjectsOfType(Ecore::EDataType)[Ecore::EDataType]->map dataToData();
}
mapping Ecore::EClass::classToClass(): Ecore::EClass{
    init{
        if(MM_Merged.objectsOfType(Ecore::EClass)->select(name = self.name)->size() = 0)then{
            result:=object Ecore::EClass{};
            result.name:=self.name;
            result._abstract:=self._abstract;
            result.interface:=self.interface;
        }endif;}
}
mapping Ecore::EReference::refToRef(): Ecore::EReference{
    name:=self.name;
    _ordered:=self._ordered;
    unique:=self.unique;
    containment:=self.containment;
    lowerBound:=self.lowerBound;
    upperBound:=self.upperBound;
    eType:=MM_Merged.objectsOfType(Ecore::EClass)->select(name = self.eType.name)->asOrderedSet()->first();
}
mapping Ecore::EAttribute::attrToAttr(): Ecore::EAttribute{
    name:=self.name;
    _ordered:=self._ordered;
    unique:=self.unique;
    iD:=self.iD;
    lowerBound:=self.lowerBound;
    upperBound:=self.upperBound;
    eType:=MM_Merged.objectsOfType(Ecore::EClassifier)->select(name = self.eType.name)->asOrderedSet()->first();
}
mapping Ecore::EDataType::dataToData() : Ecore::EDataType{
    name:=self.name;
    instanceTypeName:=self.instanceTypeName;
    serializable:=self.serializable;
}
query Ecore::EDataType::checkDataType(){
    if(MM_Merged.objectsOfType(Ecore::EDataType)->select(name = self.name)->size() = 0)then{
        MM_Merged.objectsOfType(Ecore::EPackage)->asOrderedSet()->first().eClassifiers+=self.map dataToData();  }endif;
}
query Ecore::EClass::checkClass(){
    if(MM_Merged.objectsOfType(Ecore::EClass)->select(name = self.name)->size() = 0)then{
        MM_Merged.objectsOfType(Ecore::EPackage)->asOrderedSet()->first().eClassifiers+=self.map classToClass(); }endif;
}
```

Figure 12 – QVTo for merging difference metamodels

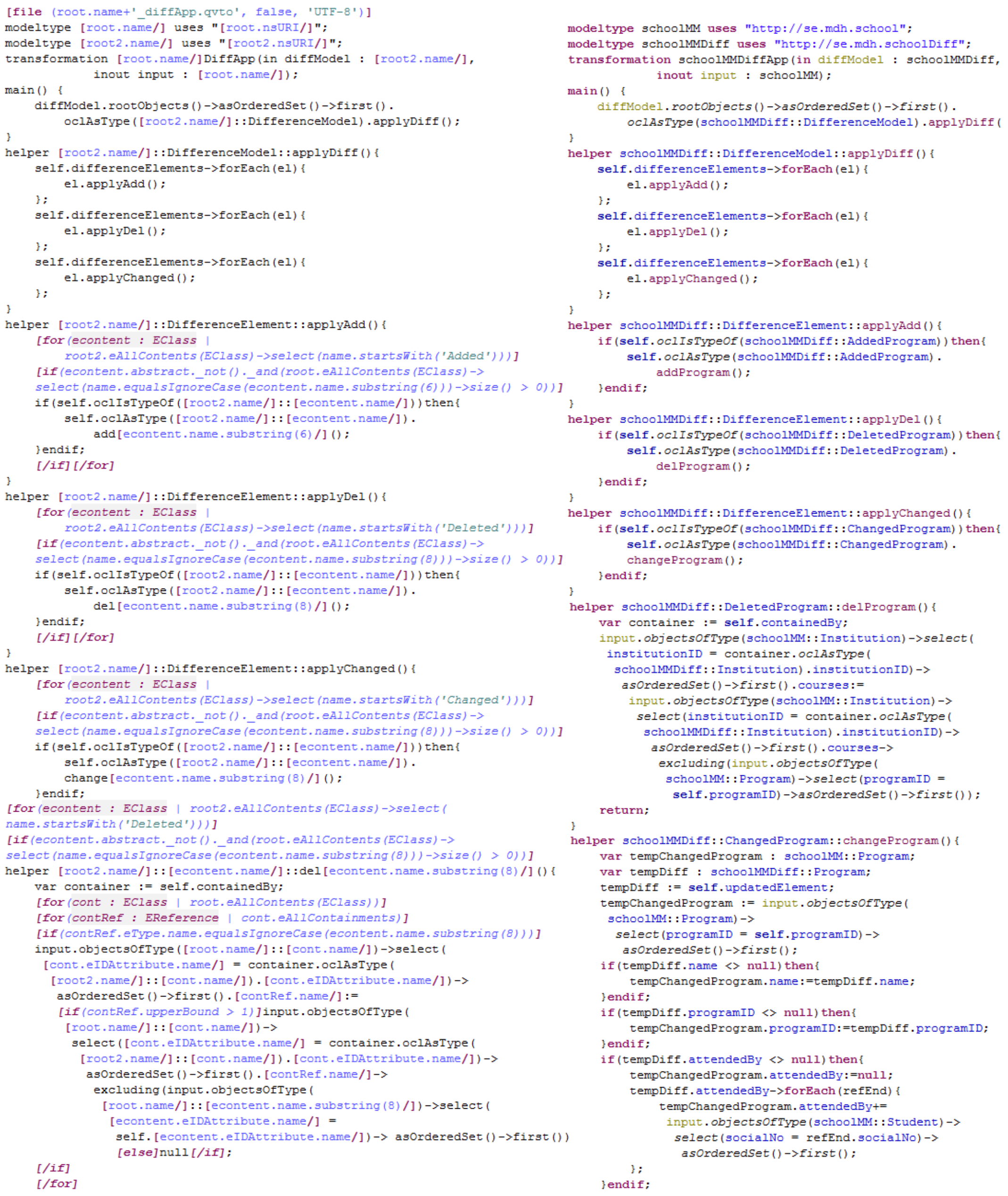
B Higher-Order Transformation for Model Differences Computation

```
[for(econtent : EClass | root.eAllContents(EClass))]
rule merge_[econtent.name/]
 merge s : left![econtent.name/]
 with t : right![econtent.name/]
 into m : diffModel![econtent.name/]{
   m.[econtent.eIDAttribute.name/] := t.[econtent.eIDAttribute.name/];
    diffModel!DifferenceModel.allInstances().first()
         .differenceElements.add(m);
   //creation of the original element
   if(s.isChanged()){
    var p = new diffModel!Changed[econtent.name/];
    p.[econtent.eIDAttribute.name/] := s.[econtent.eIDAttribute.name/];
    //check the changed attributes and references
    for(k in matchTrace.matches){
     //check attributes
     [for (attr : EAttribute | econtent.eAllAttributes)]
     if(k.info.containsKey(s.[econtent.eIDAttribute.name/]+'[attr.name/]')){
      p.[attr.name/] := s.[attr.name/];
      m.[attr.name/] := t.[attr.name/];
     }
     [/for]
     [for (ref : EReference | econtent.eAllReferences)]
     [if(not ref.containment)]
     //check references
     if(k.info.containsKey(s.[econtent.eIDAttribute.name/]+'[ref.name/]')){
     //add old ends to ChangedElement
     for(i in s.[ref.name/]){
      var oldEnd;
      //add simple element since already present in left
      if(right.allInstances().select(type|type.isTypeOf(
             right![ref.eType.name/])).exists(iter|
                  iter.[ref.eType.oclAsType(EClass).eIDAttribute.name/]==
                       i.[ref.eType.oclAsType(EClass).eIDAttribute.name/])){
       oldEnd = new diffModel![ref.eType.name/];
       //add DeletedElement since not present in right
      }else{
       oldEnd = new diffModel!Deleted[ref.eType.name/];
       //add container for added element
       var tempCont := t.copyClass();
       oldEnd.containedBy := tempCont;
       diffModel!DifferenceModel.allInstances().first().
            differenceElements.add(tempCont);
      }
      [for (attr : EAttribute|ref.eType.oclAsType(EClass).eAllAttributes)]
      oldEnd.[attr.name/] := i.[attr.name/];
      [/for]
     p.[ref.name/].add(oldEnd);
     //add old end
     diffModel!DifferenceModel.allInstances().first()
           .differenceElements.add(oldEnd);
    }
    //add new ends to new Element
    for(i in t.[ref.name/]){
     var newEnd;
     //reuse element already present in diffModel
     //add simple element since already present in left
     if(left.allInstances().select(type|
         type.isTypeOf(left![ref.eType.name/])).
           exists(iter|iter.[ref.eType.oclAsType(EClass).eIDAttribute.name/]==
               i.[ref.eType.oclAsType(EClass).eIDAttribute.name/])){
      newEnd = new diffModel![ref.eType.name/];
      //add AddedElement since not present in left
     }else{
      newEnd = new diffModel!Added[ref.eType.name/];
      //add container for added element
      var tempCont := t.copyClass();
      newEnd.containedBy := tempCont;
      diffModel!DifferenceModel.allInstances().first()
           .differenceElements.add(tempCont);
     }
     [for (attr : EAttribute | ref.eType.oclAsType(EClass).eAllAttributes)]
     newEnd.[attr.name/] := i.[attr.name/];
     [/for]
```

```
rule merge_Program
 merge s : left!Program
 with t : right!Program
 into m : diffModel!Program{
   m.programID := t.programID;
    diffModel!DifferenceModel.allInstances().first()
         .differenceElements.add(m);
   //creation of the original element
   if(s.isChanged()){
    var p = new diffModel!ChangedProgram;
    p.programID := s.programID;
    //check the changed attributes and references
    for(k in matchTrace.matches){
     //check attributes
     if(k.info.containsKey(s.programID+'name')){
      p.name := s.name;
      m.name := t.name;
     }
     if(k.info.containsKey(s.programID+'programID')){
      p.programID := s.programID;
      m.programID := t.programID;
     }
     //check references
     if(k.info.containsKey(s.programID+'attendedBy')){
     //add old ends to ChangedElement
     for(i in s.attendedBy){
      var oldEnd;
      //add simple element since already present in left
      if(right.allInstances().select(type|type.isTypeOf(
             right!Student)).exists(iter|
                  iter.socialNo==
                       i.socialNo)){
       oldEnd = new diffModel!Student;
       //add DeletedElement since not present in right
      }else{
       oldEnd = new diffModel!DeletedStudent;
       //add container for added element
       var tempCont := t.copyClass();
       oldEnd.containedBy := tempCont;
       diffModel!DifferenceModel.allInstances().first().
            differenceElements.add(tempCont);
      }
      oldEnd.name := i.name;
      oldEnd.socialNo := i.socialNo;
      oldEnd.undergraduate := i.undergraduate;
      p.attendedBy.add(oldEnd);
      //add old end
      diffModel!DifferenceModel.allInstances().first()
            .differenceElements.add(oldEnd);
     }
     //add new ends to new Element
     for(i in t.attendedBy){
      var newEnd;
      //reuse element already present in diffModel
      //add simple element since already present in left
      if(left.allInstances().select(type|
          type.isTypeOf(left!Student)).
            exists(iter|iter.socialNo==
                i.socialNo)){
       newEnd = new diffModel!Student;
       //add AddedElement since not present in left
      }else{
       newEnd = new diffModel!AddedStudent;
       //add container for added element
       var tempCont := t.copyClass();
       newEnd.containedBy := tempCont;
       diffModel!DifferenceModel.allInstances().first()
            .differenceElements.add(tempCont);
      }
      newEnd.name := i.name;
      newEnd.socialNo := i.socialNo;
      newEnd.undergraduate := i.undergraduate;
```

Figure 13 – Higher-order transformation for model differencing and resulting transformation

C Higher-Order Transformation for Model Differences Application



```
[file (root.name+'_diffApp.qvto', false, 'UTF-8')]
modeltype [root.name/] uses "[root.nsURI/]";
modeltype [root2.name/] uses "[root2.nsURI/]";
transformation [root.name/]DiffApp(in diffModel : [root2.name/],
            inout input : [root.name/]);
main() {
    diffModel.rootObjects()->asOrderedSet()->first().
        oclAsType([root2.name/]::DifferenceModel).applyDiff();
}
helper [root2.name/]::DifferenceModel::applyDiff(){
    self.differenceElements->forEach(el){
        el.applyAdd();
    };
    self.differenceElements->forEach(el){
        el.applyDel();
    };
    self.differenceElements->forEach(el){
        el.applyChanged();
    };
}
helper [root2.name/]::DifferenceElement::applyAdd(){
    [for(econtent : EClass |
        root2.eAllContents(EClass)->select(name.startsWith('Added')))]
    [if(econtent.abstract._not()._and(root.eAllContents(EClass)->
    select(name.equalsIgnoreCase(econtent.name.substring(6)))->size() > 0))]
    if(self.oclIsTypeOf([root2.name/]::[econtent.name/]))then{
        self.oclAsType([root2.name/]::[econtent.name/]).
            add[econtent.name.substring(6)/]();
    }endif;
    [/if][/for]
}
helper [root2.name/]::DifferenceElement::applyDel(){
    [for(econtent : EClass |
        root2.eAllContents(EClass)->select(name.startsWith('Deleted')))]
    [if(econtent.abstract._not()._and(root.eAllContents(EClass)->
    select(name.equalsIgnoreCase(econtent.name.substring(8)))->size() > 0))]
    if(self.oclIsTypeOf([root2.name/]::[econtent.name/]))then{
        self.oclAsType([root2.name/]::[econtent.name/]).
            del[econtent.name.substring(8)/]();
    }endif;
    [/if][/for]
}
helper [root2.name/]::DifferenceElement::applyChanged(){
    [for(econtent : EClass |
        root2.eAllContents(EClass)->select(name.startsWith('Changed')))]
    [if(econtent.abstract._not()._and(root.eAllContents(EClass)->
    select(name.equalsIgnoreCase(econtent.name.substring(8)))->size() > 0))]
    if(self.oclIsTypeOf([root2.name/]::[econtent.name/]))then{
        self.oclAsType([root2.name/]::[econtent.name/]).
        change[econtent.name.substring(8)/]();
    }endif;
[for(econtent : EClass | root2.eAllContents(EClass)->select(
name.startsWith('Deleted')))]
[if(econtent.abstract._not()._and(root.eAllContents(EClass)->
select(name.equalsIgnoreCase(econtent.name.substring(8)))->size() > 0))]
helper [root2.name/]::[econtent.name/]::del[econtent.name.substring(8)/](){
    var container := self.containedBy;
    [for(cont : EClass | root.eAllContents(EClass))]
    [for(contRef : EReference | cont.eAllContainments)]
    [if(contRef.eType.name.equalsIgnoreCase(econtent.name.substring(8)))]
    input.objectsOfType([root.name/]::[cont.name/])->select(
     [cont.eIDAttribute.name/] = container.oclAsType(
      [root2.name/]::[cont.name/]).[cont.eIDAttribute.name/])->
       asOrderedSet()->first().[contRef.name/]:=
        [if(contRef.upperBound > 1)]input.objectsOfType(
         [root.name/]::[cont.name/])->
          select([cont.eIDAttribute.name/] = container.oclAsType(
           [root2.name/]::[cont.name/]).[cont.eIDAttribute.name/])->
            asOrderedSet()->first().[contRef.name/]->
             excluding(input.objectsOfType(
              [root.name/]::[econtent.name.substring(8)/])->select(
               [econtent.eIDAttribute.name/] =
                self.[econtent.eIDAttribute.name/])-> asOrderedSet()->first())
                [else]null[/if];
    [/if]
    [/for]
```

```
modeltype schoolMM uses "http://se.mdh.school";
modeltype schoolMMDiff uses "http://se.mdh.schoolDiff";
transformation schoolMMDiffApp(in diffModel : schoolMMDiff,
            inout input : schoolMM);
main() {
    diffModel.rootObjects()->asOrderedSet()->first().
        oclAsType(schoolMMDiff::DifferenceModel).applyDiff(
}
helper schoolMMDiff::DifferenceModel::applyDiff(){
    self.differenceElements->forEach(el){
        el.applyAdd();
    };
    self.differenceElements->forEach(el){
        el.applyDel();
    };
    self.differenceElements->forEach(el){
        el.applyChanged();
    };
}
helper schoolMMDiff::DifferenceElement::applyAdd(){
    if(self.oclIsTypeOf(schoolMMDiff::AddedProgram))then{
        self.oclAsType(schoolMMDiff::AddedProgram).
            addProgram();
    }endif;
}
helper schoolMMDiff::DifferenceElement::applyDel(){
    if(self.oclIsTypeOf(schoolMMDiff::DeletedProgram))then{
        self.oclAsType(schoolMMDiff::DeletedProgram).
            delProgram();
    }endif;
}
helper schoolMMDiff::DifferenceElement::applyChanged(){
    if(self.oclIsTypeOf(schoolMMDiff::ChangedProgram))then{
        self.oclAsType(schoolMMDiff::ChangedProgram).
        changeProgram();
    }endif;
}
helper schoolMMDiff::DeletedProgram::delProgram(){
    var container := self.containedBy;
    input.objectsOfType(schoolMM::Institution)->select(
     institutionID = container.oclAsType(
      schoolMMDiff::Institution).institutionID)->
       asOrderedSet()->first().courses:=
        input.objectsOfType(schoolMM::Institution)->
         select(institutionID = container.oclAsType(
          schoolMMDiff::Institution).institutionID)->
           asOrderedSet()->first().courses->
            excluding(input.objectsOfType(
             schoolMM::Program)->select(programID =
              self.programID)->asOrderedSet()->first());
    return;
}
helper schoolMMDiff::ChangedProgram::changeProgram(){
    var tempChangedProgram : schoolMM::Program;
    var tempDiff : schoolMMDiff::Program;
    tempDiff := self.updatedElement;
    tempChangedProgram := input.objectsOfType(
     schoolMM::Program)->
      select(programID = self.programID)->
       asOrderedSet()->first();
    if(tempDiff.name <> null)then{
        tempChangedProgram.name:=tempDiff.name;
    }endif;
    if(tempDiff.programID <> null)then{
        tempChangedProgram.programID:=tempDiff.programID;
    }endif;
    if(tempDiff.attendedBy <> null)then{
        tempChangedProgram.attendedBy:=null;
        tempDiff.attendedBy->forEach(refEnd){
            tempChangedProgram.attendedBy+=
             input.objectsOfType(schoolMM::Student)->
              select(socialNo = refEnd.socialNo)->
               asOrderedSet()->first();
        };
    }endif;
```

Figure 14 – Higher-order transformation for differences application

References

[AK09] K. Altmanninger and G. Kotsis. Towards Accurate Conflict Detection in a VCS for Model Artifacts: A Comparison of Two Semantically Enhanced Approaches. In *Conceptual Modelling 2009, 6th Asia-Pacific Conf. on Conceptual Modelling (APCCM 2009)*, volume 96 of *CRPIT*, pages 139–146. Australian Computer Society, 2009.

[AP03] M. Alanen and I. Porres. Difference and Union of Models. In *UML 2003 - The Unified Modeling Language*, volume 2863 of *LNCS*, pages 2–17. Springer-Verlag, 2003. doi:10.1007/978-3-540-45221-8_2.

[ART09] ARTEMIS-JU-216682. CHESS. http://chess-project.ning.com/, February 2009.

[ASW09] K. Altmanninger, M. Seidl, and M. Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems (IJWIS)*, 5(3):271 – 304, 2009. doi:10.1108/17440080910983556.

[Bez05] J. Bezivin. On the Unification Power of Models. *Journal on Software and Systems Modeling (SoSyM)*, 4(2):171–188, 2005. doi:10.1007/s10270-005-0079-0.

[BP08] C. Brun and A. Pierantonio. Model Differences in the Eclipse Modeling Framework. *UPGRADE, The European Journal for the Informatics Professional*, April-May 2008.

[CCL11] A. Cicchetti, F. Ciccozzi, and T. Leveque. Supporting Incremental Synchronization in Hybrid Multi-View Modelling. In *Procs of the 14th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, MODELS '11. Springer-Verlag, 2011. doi:10.1007/978-3-642-29645-1_11.

[CCLP11] A. Cicchetti, F. Ciccozzi, T. Leveque, and A. Pierantonio. On the concurrent Versioning of Metamodels and Models: Challenges and possible Solutions. In *Procs of the 2nd Int. Workshop on Model Comparison in Practice.* ACM SIGSOFT, June 2011. doi:10.1145/2000410.2000414.

[CDP07] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, October 2007. doi:10.5381/jot.2007.6.9.a9.

[CDP08] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing Model Conflicts in Distributed Development. In *MoDELS '08: Procs of the 11th Int. Conf. on Model Driven Engineering Languages and Systems*, pages 311–325, 2008. doi:10.1007/978-3-540-87875-9_23.

[CDP09] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing Dependent Changes in Coupled Evolution. In *Procs of the 2nd Int. Conf. on Model Transformation (ICMT'09)*, volume 5563 of *LNCS*, pages 35–51, Zurich, Switzerland, 2009. Springer. doi:10.1007/978-3-642-02408-5_4.

[CDP10] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Model Patches in Model-Driven Engineering. In *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*, volume 6002 of *LNCS*, pages 190–204. Springer, 2010. `doi:10.1007/978-3-642-12261-3_19`.

[CH06] K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems J.*, 45(3), June 2006. `doi:10.1147/sj.453.0621`.

[CVS] CVS Project. CVS web site. `http://www.nongnu.org/cvs`.

[DIP11] D. Di Ruscio, L. Iovino, and A. Pierantonio. What is needed for managing co-evolution in mde? In *Procs of the 2nd Int. Workshop on Model Comparison in Practice*, IWMCP '11, pages 30–38, New York, NY, USA, 2011. ACM. `doi:10.1145/2000410.2000416`.

[DV07] M. D. Del Fabro and P. Valduriez. Semi-automatic Model Integration using Matching Transformations and Weaving Models. In *The 22th ACM SAC - MT Track*, pages 963–970. ACM, 2007. `doi:10.1145/1244002.1244215`.

[EPSCGM11] J. Espinazo Pagan, J. Sanchez Cuadrado, and J. Garcia Molina. Morsa: A Scalable Approach for Persisting and Accessing Large Models. In *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 77–92. Springer Berlin / Heidelberg, 2011. `doi:10.1007/978-3-642-24485-8_7`.

[Fav05] J-M. Favre. Languages evolve too! Changing the Software Time Scale. In *Procs of the 8th Int. Workshop on Principles of Software Evolution (IWPSE 2005), 5-7 September 2005, Lisbon, Portugal*, pages 33–44. IEEE Computer Society, 2005. `doi:10.1109/IWPSE.2005.22`.

[HBJ08] M. Herrmannsdoerfer, S. Benz, and E. Jürgens. Automatability of Coupled Evolution of Metamodels and Models in Practice. In *Procs of the 11th Int. Conf. MoDELS 2008, Toulouse (France)*, volume 5301 of *Lecture Notes in Computer Science*, pages 645–659. Springer, 2008. `doi:10.1007/978-3-540-87875-9_45`.

[KDPP09] D. S. Kolovos, D. Di Ruscio, R.F. Paige, and A. Pierantonio. Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing. In *Procs of the 2nd CVSM'09, ICSE09 Workshop*, Vancouver, Canada, 2009. `doi:10.1109/CVSM.2009.5071714`.

[Ken02] S. Kent. Model Driven Engineering. In *Integrated Formal Methods, 3rd Int. Conf., IFM*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer, 2002. `doi:10.1007/3-540-47884-1_16`.

[KPP06] D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Model Comparison: a Foundation for Model Composition and Model Transformation Testing. In *Procs of the Int. Workshop on Global integrated model man-*

agement (GaMMa '06), Shanghai (China), pages 13–20, New York (NY, USA), 2006. ACM Press. doi:10.1145/1138304.1138308.

[KT08] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, March 2008. doi:10.1002/9780470249260.

[Leh84] M. M. Lehman. *Program evolution*. Journal of Inf. Process. Manage., 1984. doi:10.1016/0306-4573(84)90037-2.

[LGJ07] Y. Lin, J. Gray, and F. Jouault. DSMDiff: A Differentiation Tool for Domain-Specific Models. *European Journal of Information Systems*, 16(4):349–361, August 2007. (Special Issue on Model-Driven Systems Development). doi:10.1057/palgrave.ejis.3000685.

[MRT06] MRTC. PROGRESS Webpage. http://www.mrtc.mdh.se/progress/, 2006.

[Obj07] Object Management Group (OMG). MOF Versioning and Development Lifecycle Specification, v2.0, May 2007. OMG Document: formal/07-05-01.

[OWK03] D. Ohst, M. Welle, and U. Kelter. Differences between versions of UML diagrams. In *Procs of ESEC/FSE*, pages 227–236. ACM Press, 2003. doi:10.1145/940071.940102.

[RKPP10] L.M. Rose, D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Model Migration with Epsilon Flock. In *Procs of the 3rd Int. Conf. on Theory and Practice of Model Transformations*, ICMT'10, pages 184–198, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.1007/978-3-642-13688-7_13.

[RV08] J.E. Rivera and A. Vallecillo. Representing and Operating with Model Differences. In *Procs of the 46th Int. Conf. TOOLS EUROPE 2008*, 2008. doi:10.1007/978-3-540-69824-1_9.

[SK04] J. Sprinkle and G. Karsai. A Domain-specific Visual Language for Domain Model Evolution. *Journal of Visual Languages & Computing*, 15(3-4):291–307, 2004. doi:10.1016/j.jvlc.2004.01.006.

[TDD00] S. Tichelaar, S. Ducasse, and S. Demeyer. FAMIX and XMI. In *Procs of WCRE 2000 Workshop on Exchange Formats*, pages 296–299, 2000.

[Tig] Tigris.org. Subversion version control system. http://subversion.tigris.org/.

[Tou] A. Toulmé. The EMF Compare Utility. http://www.eclipse.org/modeling/emft/.

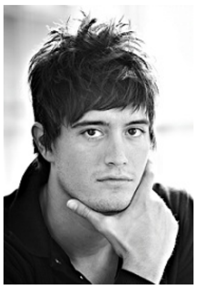
[Wac07] G. Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In *Procs of the 21st ECOOP*, volume 4069 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007. doi:10.1007/978-3-540-73589-2_28.

[WKS$^+$10] M. Wimmer, A. Kusel, J. Schönböck, W. Retschitzegger, W. Schwinger, and G. Kappel. On using Inplace Transformations for Model Co-evolution. In *Procs of the 2nd Int. Workshop on Model Transformation with ATL (MtATL 2010)*. INRIA & Ecole des Mines de Nantes, 2010.

[XS05] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *Procs of the 20th IEEE/ACM ASE*, pages 54–65. ACM, 2005. doi:10.1145/1101908.1101919.

About the authors

Antonio Cicchetti is a Senior Lecturer at Mälardalen University in Västerås (Sweden), at the Innovation, Design and Engineering Department. He is part of the Malardalen Research and Technology Centre (MRTC) and active in the Industrial Software Engineering and Model-Based Development of Embedded Systems research groups. His interests include MDE, model versioning, metamodeling, model weaving, generative techniques in Web engineering and methodologies for Web development. Additionally, he is working on the application of MDE techniques to the component-based development field, with respect to system modelling, generation of code, and Verification&Validation activities.
`antonio.cicchetti@mdh.se`.



Federico Ciccozzi is a Ph.D. student at the Institute of Innovation, Design and Engineering of Mälardalen University, Västerås (Sweden). He focuses on Model-Driven Engineering for embedded real-time systems. His interests range among MDE, CBSE, model transformations, model versioning, automatic code generation, generation and maintainance of traceability information, back-propagation of data among different artefacts (i.e., models and code) and multi-paradigm modelling. Additional interests include global software engineering and Verification&Validation activities.
`federico.ciccozzi@mdh.se`.

Thomas Leveque is a post doc researcher at Orange Labs. He made scientific contributions in the area of Model-Driven Engineering, Component-Based Software Architecture and dynamic adaption of human computer interfaces. His main research area is focused on model versioning and Model-Driven Development. His research interests also include Dynamic Service-Oriented Architecture and Real Time software.
`thomas.leveque@orange.com`.