On the concurrent Versioning of Metamodels and Models: Challenges and possible Solutions

Antonio Cicchetti
Mälardalen University, IDT
Mälardalens högskola, 72123
P.O. Box 883, Västerås,
Sweden
antonio.cicchetti@mdh.se

Federico Ciccozzi
Mälardalen University, IDT
Mälardalens högskola, 72123
P.O. Box 883, Västerås,
Sweden
federico.ciccozzi@mdh.se

Thomas Leveque
Mälardalen University, IDT
Mälardalens högskola, 72123
P.O. Box 883, Västerås,
Sweden
thomas.leveque@mdh.se

Alfonso Pierantonio
University of L’Aquila
Via Vetoio, 67100
L’Aquila, Italy
alfonso.pierantonio@di.univaq.it

ABSTRACT
Model-Driven Engineering aims at shifting the focus of software development from coding to modelling in order to reduce the complexity of realizing nowadays applications. In this respect, models are expected to evolve due to refinements, improvements, bug fixes, and so forth. Because of the same reasons, also modelling languages (i.e. metamodels) are expected to be changed, even though at a different speed if compared to models. The relevant corpus of research grown up in the latest years and dealing with both these problems considers them as separate events; however, in normal practice not all the models are migrated instantaneously due to a metamodel adaptation, rather the co-adaptation is required when commits are attempted from a local workspace to the model repository, which can demand for different management policies.

This paper illustrates the challenges arising in coping with concurrent metamodel and model versioning. In particular, it details a set of desired behaviours among which the user would usually select the appropriate management for the scenario into consideration together with entailed problems. Moreover, the work proposes corresponding solutions and discusses open issues.

Categories and Subject Descriptors
D.2.2 [Software Engineering]: Design Tools and Techniques—Object-oriented design methods; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—Re-structuring, reverse engineering, and reengineering; D.2.9 [Software Engineering]: Management—Software configuration management

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IWMC’11, June 30, 2011 Zurich - Switzerland
Copyright 2011 ACM 978-1-4503-0668-3 ..$10.00.

General Terms
Theory, Design

1. INTRODUCTION
Model-Driven Engineering (MDE) aims at facilitating the system development by creating, maintaining and manipulating abstractions of a real phenomena, i.e. models, thus reducing the complexity of the problem. This allows to focus on the aspects that most matter in the design of the application, and permit to reason about the scenario in terms of domain-specific concepts [4]. A system is developed by refining models starting from higher and moving to lower levels of abstraction; refinement is implemented by transformations over models. 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. In this respect, a metamodel describes the set of available concepts and well-formedness rules a proper model must conform to [16]. A model transformation converts a source model to a target model preserving their conformance to the respective metamodels [11].

Adopting, for instance, MDE approaches in complex software systems development processes, makes efficient versioning crucial for ensuring consistency and avoiding conflicts among modelling artefacts. Version control systems (VCSs) have been proven successful in code versioning, but they are only partially appropriate for handling versioning in the modelling domain. Differences and conflicts between different versions of a same artefact are usually detected at file-level through line-oriented text comparison which works out fine when dealing with code versioning. The same approach applied to modelling artefacts, even when taking into account their XMI-serializations, is not suitable since full homogeneity in such serialization techniques can not be ensured, even when acting in the same development environment. This may lead to erroneous detection of differences and consequently conflicts [1]. In this work we propose solutions for model versioning by acting directly to the modelling artefacts defined in their related language through model differencing and transformations as main instruments.

Even though specific techniques for model and metamodel
version management exist, they consider the two problems as separate from each other. However, in real-life version management it is needed to support these concurrent evolutions, since manipulation rates of models and metamodels are often different and not always aligned.

This work discusses motivations, challenges and possible solutions for versioning of modelling artefacts, in particular when 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 all cases well-formedness issues arise, since neither intended modifications on the previous version of the language could be completely valid in the new version, nor manipulations operated by means of the old language could be legal in the new version.

The remainder of this paper is structured as follows. Section 2 identifies 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 and arising challenges in language evolution that we considered, while the proposed solution and its implementation is described in detail in Section 4. A discussion on the issues that remain still unsolved is presented in Section 5. Eventually, conclusions and possible future work conclude the paper in Section 6.

2. BACKGROUND

In the history of software development, maintenance activities have been recognized as unavoidable from the early stages [19], thus techniques have been introduced and enhanced to deal with the management of source code evolution. In this respect, state-of-the-art tools provide a repository, either centralized or distributed, in which the current version of the application is stored. Then, in order to allow a more efficient process, concurrent development is controlled in an optimistic way; each developer has a local workspace on which her/him performs modifications that later on are committed to the repository in order to update the current version and advertise it to the other developers. Therefore, locking mechanisms can be avoided (even if still allowed when necessary), while the eventualities of conflicts is left open. In fact, whenever other developers want to commit their own changes as well, they will need first to update their local workspaces to re-align their local version to the shared one, merge their manipulations and the ones already committed also fixing possible divergences between them, and subsequently commit the resulting revision back to repository [3].

With the growing adoption of MDE in complex software development the need of appropriate support for versioning management has become one of the main topics of the current research in the field. In the following, such efforts for achieving corresponding solutions are described in order to better clarify problems, available solutions, and the context in which the contribution of this paper is located.

2.1 Model versioning

The need of managing versions of models arose as soon as they were introduced in normal practices of software development. In fact, MDE promotes them as first class citizens exploited to generate code, perform analysis of system’s properties, and so forth [4]. Initial attempts to deal with model versioning relied on the available expertise on text-based mechanisms like Subversion [27] and CVS [10]: models are serialized in a more-or-less structured format ranging from plain text to XML documents, and thus their versioning is reduced to source code management. However, such solutions suffer an abstraction level mismatch, i.e. text-based techniques cannot grasp manipulations rationally at modelling level [1, 17]. Consequently, a number of techniques have been introduced for detecting, storing, and visualizing [5] the evolution at modelling level. Some of these approaches are specific for a certain modelling language as the UML, like the solutions proposed in [22, 30]; others are language independent and can be applied to any kind of models [20, 7, 23]. Additional mechanisms store model histories through snapshots at corresponding points in time of the development life-cycle [26, 21].

The techniques described so far can be considered as the foundation on top of which model repositories are built: in general, the latest version of the model is stored completely to be ready to use, whereas each of the past revisions is memorized in terms of differences with respect to its successor (or analogously predecessor). In this way the repository size can be optimized since usually the amount of changes is considerably smaller than the number of elements existing in the whole model. 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 [2, 9]. For an extensive survey on model versioning tools, related methodologies, and raising difficulties the reader is referred to [3]; for the purpose of this paper 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

Embracing the MDE vision entails the use of models in all aspect of the software development life cycle; often Domain-Specific Models are exploited to provide a better abstraction level by using concepts closer to the domain experts and to improve automation of derived artefacts as analysis, testing, and code [4, 15]. Therefore, typically modelling languages are designed in an incremental process in which expressiveness is manipulated to add, fix, and/or refine available concepts [13]. Since metamodels are models themselves, the aforementioned techniques dealing with model versioning can be transferred also to metamodelling settings [6]. However, whenever a metamodel evolves, corresponding migrations have to be operated since existing model instances could not be well-formed any more [25]. In this respect, latest years have shown a growing number of research works approaching the problem by means of different solutions ranging from manual co-adaptation [24, 25], to semi-automatic migration through re-use of recurring strategies [14, 28], to automatic co-evolution directly generated from the manipulations made to the metamodel [8, 12].

Analogously to model versioning, it is worth noting that
metamodel evolution and the corresponding model co-evolution are managed as happening instantaneously.

2.3 Contribution

As discussed so far, both model and metamodel versioning have been widely recognized as relevant problems demanding for adequate support in order to improve MDE acceptability in industrial software development. Nonetheless, these issues 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 both 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 paper is based on the assumption that the language is explicitly versioned, that is 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 one is kept as the (meta)model as a whole [7].

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 $\text{MM}_a$ is checked out through an update operation from the repository into the local workspaces together with the current version of the model, i.e. $\text{M}_a$ (shown on the left side) \(^1\).

Metamodel and model manipulations happen at different rates with respect to the time line of the development process: in fact, after a major stabilization of the language in which metamodel refinements can be more frequent, model mutations 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 $\text{M}_a$ is firstly modified in $\text{Workspace}_2$ and committed as the revision $\text{M}_b^3$ into the repository. In the meantime, in $\text{Workspace}_1$ some modifications occur too (see $\text{M}_a^5$): when the developer tries to commit her/his version, $\text{M}_b^3$ is retrieved from the repository by means of an update in order to be merged with the local version, possible conflicts are solved, and the resulting version is committed back into the repository as revision $\text{M}_b^5$.

The evolution pattern drawn so far can proceed smoothly enough until a metamodel adaptation happens: in fact, some users could be still editing their models by means of the old version of the language, hence modifications can become inconsistent with respect to the current language. For instance, in Figure 1 an evolution is operated on $\text{MM}_a$ in order to obtain $\text{MM}_b$: while $\text{M}_b^3$ in the repository is migrated to $\text{M}_b$, to restore its well-formedness with $\text{MM}_b$, in the local workspaces there exist pending updates performed as conforming to $\text{MM}_a$, i.e. $\text{M}_a^5$, $\text{M}_b^5$. As a consequence the following issues arise: should developers first try to migrate local models to $\text{MM}_b$ 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 $\text{MM}_b$? For instance, while $\text{M}_b^3$ has been obtained from the latest version before the migration (i.e., $\text{M}_a^5$, $\text{M}_b^5$), “missed” modifications made within $\text{Workspace}_1$ and updating directly from $\text{M}_b^3$ could cause loss of information.

At this point it is worth noting that, as explained better later on in the paper, in real life practice developers will 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, just to mention a few. Of course, an ideal development process would entail a synchronized environment in which all the developers are using the same set of tools and the same version of each of them. Unfortunately, in real-life 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 4 we discuss possible solutions to deal with those challenges correspondingly.

---

\(^1\)For the sake of readability conformance relations have been omitted and they are given implicitly through the model name.
a language which is formally defined in terms of a metamodel. In fact, as source code files are compatible 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 (i.e. metamodel) in terms of versions, and consequently models conforming to them, is a challenging task that is crucial in order to maintain a fully synchronized and consistent modelling environment.

In the following, we describe three scenarios in which the challenges related to language evolution that are going to be treated in this work 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 updated in the user workspace (see Figure 8); 
- **Scenario 3**: A model version designed according an old metamodel version is retrieved while keeping the latest metamodel version as actual in the user workspace (see Figure 9).

In Scenario 1 (Figure 2), concurrent modifications of a model $M_a$ are performed in conformance to metamodel $MM_a$ as starting from parallel model versions updated in Workspace1 and Workspace2. A new metamodel version $MM_b$ is then created and updated to Workspace1; then, $M_a$ is co-adapted to conform to it and $M_b$ is obtained. $M_b$ is modified and subsequently committed to the repository and such operation leads to the creation of version $M_{1b}$.

Modifications represented by $M_{1a}$ must be saved without updating metamodel version in Workspace2 to the latest available in the repository mainly for two reasons: (i) forcing the usage of $MM_b$ 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.

In order to better explain this scenario Figure 3 illustrates the initial version of $M_a$ as conforming to the metamodel $MM_a$, a simplified UML class diagram\(^2\) shown in the upper part of Figure 4. Then, an evolution of a simplified UML class dia-

\(^2\)Both the metamodel and its evolution have been inspired from [29] and redefined in Ecore for this work.
classes while the specializations of attribute disappeared to be managed through the id attribute. As a consequence, $M_a$ has to be migrated in order to be well-formed with respect to the new language as described in Figure 5.

At this point some changes are operated on $M_a$ in Workspace2 to obtain $M_1^a$ as depicted in Figure 6: the existing Employer class has been renamed as Company and a new attribute Address has been added to such class. In the meanwhile, manipulations are done on $M_b$ in Workspace1 too, where the current metamodel is $MM_b$, giving place to the model $M_1^b$ shown in Figure 7: class SocialNumber has been renamed to IdentificationNumber; moreover, a new association with role subscribe has been added between Employment and the new class Contract. It is worth noting that both the changes made on $M_a$ to obtain $M_1^a$ and $M_b$ to obtain $M_1^b$ conform to their corresponding metamodels, i.e. $MM_a$ and $MM_b$; in this respect, some modifications make sense for both the models, while others are specific of the version of the language.

For instance, the addition of subscribe association end in $M_1^b$ has no correspondence in $M_1^a$ since association classes do not exist in $MM_a$. Therefore, by recalling Scenario 1, modifications made on the repository can not be fully updated to Workspace1 and vice versa, changes made to the local version of $M_a$ can not be committed to the repository.

In Scenario 2 (Figure 8) we suppose that there are no issues that prevent the update of the metamodel version; therefore $M_1^a$ and $M_1^b$ can be merged after the metamodel $MM_a$ in the Workspace2 is updated to version $MM_b$. By taking into account the example described above, $M_1^a$ and $M_1^b$ have to be adequately merged and later on the migration of the local version in Workspace2 can be operated. After such migration Workspace2 and the repository will be using the same metamodel version ($MM_b$), thus allowing to use the normal versioning procedures.

Challenges do not only arise when going forward to newer versions of the metamodel. In fact, consistency may be undermined also when the user tries to go backwards to a previous version of a model which conforms to an old metamodel revision, while trying to keep the current metamodel version in the workspace (Figure 9). 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 file based Version Control System (VCS), model VCS save only differences between versions in order to reduce the repository size, hence 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 are in a form which is metamodel specific (see Section 2). As a result, model difference representations depend on the metamodel version taken into account. Therefore, retrieving 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 remains an open issue, even though some solution has been proposed, as explained later on in this work.

The next section presents a proposal to solve the challenges described so far in the three detailed scenarios: it relies on a precise representation of model and metamodel differences and on a technique to merge manipulations pertaining to different languages in order to allow their management.
4. PROPOSED SOLUTION

As discussed so far, challenges arise when managing the concurrent evolution of models and metamodels; therefore, let us suppose that on the repository there is the last version of the model as migrated by the co-evolution transformation. Moreover, in the workspaces there exist delta 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 adaptations and model manipulations. It is worth noting that the latter ones also include model migrations, since they can either be generated as the co-evolution strategy to re-align models with the new version of the metamodel, or they can be retrieved at the end of the co-adaptation process if they are carried out manually or through automated mechanisms (see Section 2).

4.1 Model difference representation

For both metamodel and model versioning we adopt a model-based representation to store the differences between an old and a new language (or model) revision. Such representation is based on an existing work [7] 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 [6]; 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. Moreover, we will consider the difference calculation as provided and working properly, i.e. correctly detecting model changes 3.

A difference language is obtained through the automated extension of the metamodel the models conform to: for instance, developers working with metamodel \( \text{MM}_b \) will modify models and produce new instances conforming to the language depicted in Figure 4, as described in Section 3. Therefore, the language able to represent model manipulations will be generated by means of an appropriate extension of \( \text{MM}_b \), called \( \text{MM}_bD \), an excerpt of which is shown in Figure 10. By going into more details, each non-abstract metaclass \( MC \) in the metamodel induces three specializations, namely \( \text{AddedMC} \), \( \text{DeletedMC} \), and \( \text{ChangedMC} \), which are exploited to represent the additions, deletions, and changes, respectively, of such metaclass. Moreover, an \( \text{updatedElement} \) association connects an updated metaelement with its corresponding new version 4. Therefore, the \( \text{Class} \) metaclass in \( \text{MM}_b \) induces the creation of the corresponding \( \text{AddedClass} \), \( \text{DeletedClass} \), and \( \text{ChangedClass} \), the \( \text{Attribute} \) metaclass induces \( \text{AddedAttribute} \), \( \text{DeletedAttribute} \), and \( \text{ChangedAttribute} \), and so forth. In the same way, \( \text{MM}_a \) metamodel is extended to obtain the corresponding \( \text{MM}_aD \) difference language 5. All the possible changes a model conforming to \( \text{MM}_b \)

---

3 Model differencing is a well known issue in the model versioning area and it goes far beyond the scope of this paper; for a deeper discussion on the topic the reader is referred to [18].

4 For further details on the representation approach and the motivations underlying some design choices the reader is referred to [7].

5 Analogously, an appropriate extension of the Ecore metamodel is generated to represent revisions of metamodel, e.g. to store differences between \( \text{MM}_a \) and \( \text{MM}_b \). However, for the sake of space both that metamodel and \( \text{MM}_aD \) are omitted. For more details the reader is referred to [6].
may be subject to can be represented through the difference metamodel $M_{D}$ illustrated above. For example, in Figure 11 an excerpt of the $M_{b}$ manipulation, whose result $M_{b}'$ is visible in Section 3, Figure 7, is stored by such difference language. In particular, in the upper part of the picture changes to the class $\text{Person}$ are represented: $\text{SocialNumber}$ attribute is renamed to $\text{IdentificationNumber}$ while $\text{Age}$ is deleted. Moreover, in the lower part of the figure a new reference end with the role $\text{subscribe}$ is added between the new $\text{Contract}$ class and the existing $\text{Employment}$ association. As discussed in [7], the difference 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 scenario described in Section 3, manipulations made on the repository and the ones locally operated pertain to different languages, i.e. $M_{b}'_{aD}$ and $M_{b}'_{bD}$, respectively. In this respect, we propose a general solution based on a metamodel merging technique able to produce a union metamodel including concepts from both the versions. It is worth mentioning that this comparability problem is quite common as emerging when facing metamodel evolution and trying to represent the modifications of the migrated models in some way, and relaxing language constraints to build-up a sort of lingua franca is an accepted practice [14, 29]. In the following, we first describe the mechanism to unify 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 carried by the scenarios presented in Section 3.

### 4.2 Merging and Splitting 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 $M_{a}'_{aD}$ and $M_{b}'_{bD}$) that result into new merged difference metamodel $M_{bD_{aD}bD}$ depicted in Figure 12 which is capable to represent all the information conforming to both $M_{a}'_{aD}$ and $M_{b}'_{bD}$. That is to say that the new metamodel will be able to represent a joint difference model (Figure 13) which is the merged version of the single difference models (i.e. $M_{a}'_{aD}$ and $M_{b}'_{bD}$). The algorithm for the creation of $M_{bD_{aD}bD}$ resembles the one proposed in [29] and consists of two main steps:

1. **Identification and merge of mergeable metaclasses**: In this first step, the classes of $M_{a}'_{aD}$ and $M_{b}'_{bD}$ are compared and, if they have same name, merged into one single class in $M_{bD_{aD}bD}$. The new class will get the union of the features (i.e. attributes and references) and superclasses of both $M_{a}'_{aD}$ and $M_{b}'_{bD}$; 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 $M_{a}'_{aD}$ and $M_{b}'_{bD}$**: The metaclasses in $M_{a}'_{aD}$ that do no have a correspondent in $M_{b}'_{bD}$ and vice versa are directly included to $M_{bD_{aD}bD}$ with their own features.

The union metamodel is now ready to enable the representation of a merged information, that we will call $M_{lm}$ coming from two difference models conforming to the two merged metamodels $M_{a}'_{aD}$ and $M_{b}'_{bD}$. The result of merging information from the difference models $M_{a}'_{aD}$ (Figure 6) conforming to $M_{a}'_{aD}$ and $M_{b}'_{bD}$ (Figure 7) conforming to $M_{b}'_{bD}$ is therefore conforming to $M_{bD_{aD}bD}$, as depicted in Figure 13. In order to be able to keep trace of which information came from which source difference metamodel, the metaclasses introduced in $M_{bD_{aD}bD}$ are annotated as: (i) merged, if coming from a merging of metaclasses, (ii) mma if included from a non-merged metaclass of $M_{a}'_{aD}$, and (iii) mmb if included from a non-merged metaclass of $M_{b}'_{bD}$. This additional in-

---

**Figure 11**: An excerpt of the $M_{b}$ evolution represented by the difference metamodel.

---

**Figure 12**: An excerpt of the union metamodel resulting from the merge of $M_{a}'_{aD}$ and $M_{b}'_{bD}$. 

---
Figure 13: Example of difference models merging

formation is needed for enabling the next task of splitting, to be performed for using the modifications represented in the merged difference model Figure 13 to align, as in Scenario 1, the evolved version of \( \mathcal{M}_a \) (i.e. \( \mathcal{M}_a^1 \)) in Workspace2 to the evolved one (\( \mathcal{M}_b^1 \)) in Workspace1. In fact, there could be changes that, even if performed on a later metamodel version \( \mathcal{M}_b \), have still a meaning under \( \mathcal{M}_a \), and therefore can be applied to \( \mathcal{M}_a^1 \).

Once obtained the merged modifications as coming from both the local revisions and the repository, the developer can verify the operations and possibly fix some conflicts. At the end of this process the manipulations are ready to be sent back to the corresponding sides and in a format conforming to the appropriate languages (i.e. \( \mathcal{M}_a \) and \( \mathcal{M}_b \)). In order to enable such operation, the merged differences have to be split depending to the language they pertain to; in other words, a split operation is needed such as it can invert the merging described above.

The first step of the splitting task consists in identifying the modifications in \( \mathcal{M}_{dm} \) that can be applied to \( \mathcal{M}_b^1 \); the annotations present in \( \mathcal{M}_{dm} \) allows to identify such eligible modifications by looking the metaclasses annotated with either merged or mma. In fact, being \( \mathcal{M}_a^1 \) conform to \( \mathcal{M}_b \), modifications that only concern \( \mathcal{M}_a \) cannot be applied to it (that is the ones annotated with mmb). Once identified the valid modifications, thanks to the notation introduced by the difference metamodels it is possible to recognize which elements have been created, deleted or updated so that the equivalent modification can be applied on \( \mathcal{M}_a^1 \).

4.3 Solving the challenges

Scenario 1.

This is the most complex scenario since developers cannot upgrade their metamodel to the current version, therefore they will continue working (and hence modifying) models by means of the previous metamodel. In this respect, the problem of incompatibility between manipulations operated through the different metamodels can be tackled by the metamodel merging technique explained above. In particular, \( \mathcal{M}_{\text{Diff}} \) metamodel shown in Figure 12 includes concepts from both the languages and can support the representation of changes concurrently made both on the repository and locally (see Figure 13).

By means of \( \mathcal{M}_{\text{Diff}} \) concurrent models can be compared at the same level, appropriate merge operations performed, and then the resulting version can be committed to the repository by the splitting operation clarified in the previous section. It is important to notice that the union metamodel is exploited only for merge purposes, whereas for editing the model on the repository and the one in the user’s workspace corresponding metamodel versions are used. This is a fundamental constraint in order to ensure the well-formedness of modifications operated on both sides.

Scenario 2.

In this setting developers’ choice is to upgrade their changes to the current version of the metamodel and then try to merge her/his modifications with the latest revision on the repository. The easy part of this scenario is that this step has to be performed only once, for each pending manipulation operated as conforming to an old version of the metamodel. In this case, the migrated model version on the repository can be compared against the revision operated in the developer’s workspace through the union metamodel (i.e. \( \mathcal{M}_{\text{Diff}} \)), perform the required merge fixes, and then commit the obtained revision to the repository, which will also be the current version adopted in the workspace. In this respect, after the well-formedness of the user’s workspace has been restored by means of a migration toward the current metamodel revision, the usual versioning approach can be exploited.

Scenario 3.

This case is useful whenever it is needed to retrieve past versions, even if conforming to a different modelling language. For example, software houses typically develop applications through the support of frameworks that offer a basic set of services on which build-up the system. If the language is evolved, developers will need to pick a past version and migrate it in order to be used in the current environment. In this respect, the metamodel merging technique illustrated above allows to navigate the modifications back to the desired version, even if model migrations happened; in fact, the necessary manipulations can be stored in an appropriate union metamodel and reproduced when needed to restore the desired model revision.

5. DISCUSSION

In the previous section have been illustrated possible solutions to the issues presented in Section 3. However, some problems still remain open: in particular, when considering
Scenario 1 the proposed approach allows the existence of parallel versions developed as conforming to different languages. It is worth to notice that while consistency can be guaranteed for the version committed on the repository (and hence for all the local modifications conforming to the same metamodel, i.e. $MM_0$), the same can not be ensured for local revisions conforming to old metamodels (e.g. $MM_i$). In fact, in this case local changes could have no counterpart in the new metamodel and hence stay hidden on the developer side. In turn, other developers working with the same language revision ($MM_i$) would not get such updates from the repository since there only modifications pertaining to $MM_0$ are visible, thus causing inconsistencies between those versions. Nonetheless, whenever the developers would evolve to the new language version, such inconsistencies would be fixed and all the local copies would become consistent with the revision stored on the repository. As expected, there could be some loss in the migration process: in particular, if $MM_i$ is obtained by deleting some entities from $MM_0$, current modifications made locally could disappear when migrating to the newer metamodel version.

A possible alternative solution to the union of metamodels to support the three different scenarios could be to create a migration bridge able to upgrade (downgrade) $^6$ a model each time it is committed (updated) to (from) the repository. In particular, each time an update is required from the repository, an inverse transformation has to be provided such that the current revision can be downgraded to the previous version of the metamodel. In this way, the user can merge the portion that makes sense in her/his metamodel version and then commit her/his changes back to the repository. As expected, also in this latest step a migration is required to re-align the model to the metamodel version on the repository. However this solution poses some technical difficulties: in order to be sure that migration steps are correct the migration transformation has to be consistently generated and reproduced each time it is needed, that is it has always to perform the same co-evolution operations with respect to the same evolution of the metamodel; moreover, whenever user’s input would be needed in the migration phase, it has to be stored and replicated when the co-evolution is re-applied; finally, the downgrading transformation has to be the exact inverse mapping of the upgrading one.

Regardless the approach exploited to support concurrent model and metamodel evolution management, the scenarios presented in this work can be considered as general problems that have to be faced. By embracing the MDE vision in which everything is a model [4], an exhaustive discussion on parallel versioning of models and metamodels could have also included, for instance, model transformations or editing tools, since they rely on the current language revision and hence co-evolve whenever the metamodel is evolved. However, it is worth noting that even in such cases what could change is the strategy exploited to manage concurrent modifications, i.e. whether to make different metamodel versions to co-exist or if a migration to the new revision of the language would be always desired; in any case we would fall in the scenarios illustrated in this work. In particular, on the one hand for model transformations development would appear more appropriate to migrate as soon as possible to the new version of the language, since what is going to be defined could be erroneous, or even unusable. On the other hand, a metamodel adaptation in the middle of a tool design process would be rarely integrated immediately, but the achievement of some milestone would be waited for before trying to co-adapt the current artefacts.

6. CONCLUSIONS AND FUTURE WORK

This work discussed the problem of managing concurrent revisions of models and metamodels. Typically these issues are considered as separate, i.e. either model or metamodel versioning and model migration concerns are taken into account; whereas, we claimed that in normal development practice they interfere each other thus demanding for adequate support. In this respect, this paper illustrated a set of challenges arising when dealing with such a problem and in particular with scenarios where: i) users wish to commit manipulations made by the old version of the language directly to the repository since they make sense only as they are. For each scenario, corresponding solutions have been described to support parallel adaptations of metamodels and models in a distributed development environment. The proposed approach relies on the concept of a union metamodel that enables the contemporary representation of manipulations coming from both old and new language versions and hence their management in terms of change commits, conflict detection and resolution.

For future works we plan to validate our proposal against an industrial case study, in order to discover both possible pitfalls in the creation of the super metamodel and in the management of manipulations conforming to different languages versions. Moreover, we will investigate on a better management of inconsistencies between workspaces adopting different language revisions as discussed in Section 5.

7. REFERENCES


