UNIVERSITÀ DEGLI STUDI DI PARMA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

UNA TECNICA PER IL 3-WAY MERGE DI MODELLI BASATA SU XMI

AN XMI-BASED TECHNIQUE FOR THE 3-WAY MERGE OF MODELS

Relatore

Dott. Ing. F. BERGENTI

Correlatore

PROF. L. BENDIX

Tesi di Laurea di

ANTONIO MARTINI

Ad Ildikó ed alla mia famiglia

Ringraziamenti

Desidero innanzitutto ringraziare il Professor Federico Bergenti ed il Professor Lars Bendix per i preziosi insegnamenti durante i due anni di laurea magistrale e per le numerose ore dedicate alla mia tesi. Vorrei inoltre ringraziare il professor Lars Bendix per aver esteso ben oltre i confini istituzionali del "Consigliere Scientifico" e del “Correlatore” il suo appoggio al mio lavoro. Intendo poi ringraziare il dipartimento di informatica dell’Università di Lund per avermi fornito le strutture necessarie, i testi ed i dati indispensabili per la realizzazione della tesi. Desidero ringraziare l'Università di Parma per avermi dato l'opportunità di usufruire di una borsa di studio per il soggiorno a Lund, in Svezia, durante il quale è stato svolto gran parte di questo lavoro. Inoltre, vorrei esprimere la mia gratitudine al Dr Maximilian Koegel presso l'Università di Monaco di Baviera per i numerosi consigli durante la fase di ricerca. Infine, ho desiderio di ringraziare con affetto i miei genitori e la mia sorella Francesca per il sostegno ed il grande aiuto che mi hanno dato. Un ringraziamento in particolare ad Ildikó per essermi stata vicina ogni momento durante questo lavoro.

Indice

Indice delle Figure

Prefazione

Negli ultimi anni è incrementata la realizzazione di modelli per lo sviluppo di software, sia per la sua rappresentazione astratta, che per la creazione automatica del codice, divenendo un passaggio critico nella fase di progettazione. Molti editor sono stati creati per supportare la creazione grafica di modelli. Tuttavia, poco è stato fatto per permettere una valida interazione tra modelli provenienti da strumenti differenti, inficiando ogni attività di riconoscimento, confronto e fusione (*merge*) tra di essi. Infatti, come tutti gli artefatti utilizzati durante lo sviluppo di un software, anche i modelli possono subire variazioni, a volte anche simultanee, da parte degli sviluppatori e sono perciò spesso gestiti da tool di versionamento. Il fatto di non poterne automatizzare il riconoscimento, il confronto e la fusione, provoca l'abbondanza di lavoro manuale spesso difficoltoso e generatore di errori.

I problemi sono dovuti principalmente al passaggio dallo sviluppo orientato al codice a quello orientato ai modelli: infatti, per la prima tipologia si possono trovare un nutrito insieme di strumenti software ben funzionanti per la gestione del merge (fusione) testuale, i quali però non risultano essere altrettanto utili nel merge dei modelli. Questo perché i modelli sono serializzati utilizzando lo standard XMI, un particolare dialetto XML che consente il salvataggio fisico di documenti contenenti dati strutturati. La scelta di analizzare e confrontare semplicemente le varie linee testuali non risulta essere più adatta per tale tipologia di documenti. Si possono trovare alcuni nuovi strumenti per il merge automatico di modelli, ma anch'essi non risultano essere abbastanza precisi, o risultano essere legati ad uno specifico linguaggio (o più spesso ad una versione di esso). Inoltre, hanno tutti la peculiarità di essere interattivi: questo, nonostante sia ritenuto solitamente essere un vantaggio per l'utente, nel processo di model-merge risulta invece creare dei problemi. Tale sistema costringe lo sviluppatore a seguire l'intero processo (che a volte può anche rivelarsi molto lungo) secondo un “percorso” forzato dal merge-tool. Infatti risolvere un conflitto dopo l'altro non è producente, in quanto spesso essi sono collegati tra loro e necessitano di essere analizzati contemporaneamente. L'utente sarebbe più spesso portato a creare una nuova soluzione “ad hoc” che integri tutti i cambiamenti apportati simultaneamente piuttosto che risolverne uno a uno (senza contare che le soluzioni individuali possono a loro volta creare nuovi conflitti).

In questa tesi, l'obiettivo è quello di identificare e studiare la realizzabilità di un processo automatico che realizzi il merge di due modelli utilizzando i file XMI generati da un qualsiasi editor. Tali modelli rappresentano le due versioni ottenute dalla modifica di uno stesso modello detto common ancestor. Per questo si parla di 3-way merge, nel quale è possibile sfruttare le informazioni contenute nel modello di base da cui sono derivati i file deputati alla fusione. Presentiamo un processo che utilizza solo l'informazione derivata dalla sintassi XMI ed è suddiviso in cinque passi, ognuno dei quali risulta analizzabile e sviluppabile indipendentemente. Tuttavia, è d'obbligo precisare che l'output fornito da ognuno di essi deve essere poi utilizzato come input per il passo successivo. Si è quindi ipotizzato un algoritmo astratto che rappresentasse i passi descritti precedentemente. L'output di questo procedimento risulta essere un nuovo file XMI. Tale file non può rappresentare direttamente il modello finale: è stata preferita la rappresentazione organica dei problemi in modo da fornire tutta l'informazione per la realizzazione del merge da parte dello sviluppatore, piuttosto che l'automatizzazione del merge stesso. Questo per evitare l'inevitabile perdita di informazione che si avrebbe con l'applicazione automatica di alcuni cambiamenti (ad esempio quelli che creano conflitti). I risultati ottenuti sono stati discussi e si è riscontrata la necessità di evidenziare alcune restrizioni sotto le quali tali risultati mantengono la loro validità. I problemi principali sono legati ad una mancanza di omogeneità tra documenti XMI derivati da diverse versioni del linguaggio stesso, nonché da diverse interpretazioni dei vari editor.

Infine, si è implementato in Java un nostro software (XMIMerge) che dimostrasse l'applicabilità di alcuni principi teorici mostrati precedentemente. Tale strumento esegue i primi tre passi del processo di merge, con l'aggiunta di una rappresentazione grafica in grado di mostrare indipendentemente tutti i problemi e gli elementi che li hanno causati. In questo caso si parla di “virtual merge”, piuttosto che di merge. Tale soluzione evita i problemi di interattività e permette all'utente di verificare contemporaneamente su un solo pannello di visualizzazione i tre modelli, consentendo l'analisi dei possibili problemi a cui si andrebbe incontro nella realizzazione di un ipotetico merge.

Chapter 1

Introduction

Parallel working among several developers gives many advantages in a software development process, but it causes also problems: among them, as Babich says [3], there is double maintenance. To avoid this problem, developers often have to integrate their works with the latest version to be able to release their own version which includes the previous changes as well. This work is called **merging process**: the developer mainly has to find changes among his own version, the last version on the repository and, in case, the common ancestor. Often, his changes conflict with those added by others, so these conflicts have to be resolved. This task (the merging process) is quite important and hard to handle, so it should be carried out frequently and carefully [9]: consequently, to be well performed, it requires a set of tools.

In the code centric development, we find a lot of good text-based tools which help managing the merge task. Lately, industries are increasing the use of Model Driven Development, creating models to auto-generate code. Nevertheless, the environment is not yet mature enough to support adequately the parallel work of different developers. Unfortunately, moving from a code centric development strategy to a model centric one showed that former textual-based merge tools do not work appropriately with models [5]. In fact, models are serialized using the standard XMI: a language which creates documents containing structured data. Therefore, the comparison of text lines is not the best choice anymore, as a little change at the syntax and semantic level could correspond to several changes on the text level. Consequently, we need a more sophisticated solution in order to find, compare and resolve conflicts between model files, changing for example the granularity of the unit of comparison [13] from the text line to the node of a tree. Model merge tools are not precise enough either, since they have some problems such as detecting too many false positives and false negatives, or perform the merge without taking in consideration the smallest possible element [7], but just raising a conflict if the same top level object is modified (too coarse granularity of the unit of comparison). Moreover, they are all oriented towards interactivity, which means that the developer has to follow the entire merge process, conflict by conflict. Furthermore they have to choose "on the fly" among (probably) wrong alternatives provided, instead of looking for the connections between them, creating an "ad hoc" solution [8].

The aim of this thesis is to investigate the feasibility of a merge process for models using only the XMI serialization. We take three XMI files representing three models (the common ancestor and the two changed versions) and we provide a new file representing a merged XMI. First of all, the merge algorithm should find all the changes and should detect the highest possible number of conflicts among them (in order to avoid false negatives), but it should also detect conflicts "to the bottom", which means that there is a conflict when the same smallest possible thing is changed (in order to avoid false positives). Moreover, we would like to represent the information about all changes of both modified versions in the merged file. So all the non-conflict changes have to be present and highlighted in the merged file. In case we had a conflict between two changes, it could be resolved by ignoring one of them: in such a situation, we would like to know which change was ignored and why. In case we had an unsolvable conflict, we should represent both possible alternatives in the merged file.

In the following sections, we will explain in details our context with respect to the model merge problem, then we will deal with the characteristics of the XMI language, its structure, problems and advantages when used to perform a model merge algorithm (chapter 2). In chapter 3 we will describe the requirements of a correct merge and we will explain our proposal of a 5-step merge process. Then, we will show an abstract algorithm to be implemented (chapter 4). In chapter 5 we describe our Java implementation of a tool which provide a virtual merge for models serialized by ArgoUML [1]. Finally, we will discuss our results, comparing our work with other related ones and presenting ideas to improve the work and directions for further research (chapter 6).

Chapter 2

Background and context

In this chapter we contextualize our work presenting the general problem of versioning and merging models. Then we list the approaches used and we explain why we chose them over other existing solutions. Moreover, we introduce XMI showing its basic role in the model serialization. Finally, we provide some details about serialization patterns used by XMI in order to motivate some subsequent assumptions, and to make the followings more comprehensible.

2.1 Context

Lately, models are widely used both to design a product and to auto-generate code in industries with the increasing use of Model Driven Development. Another powerful strategy in software development is the parallel work of many developers, but it presents some drawbacks which have to be handled: especially, the problem that Babich called **double maintenance** [3]. As we can see in figure 1, two developers have simultaneously modified the same version of an artifact: in this case, one of them has to commit his version on the shared repository, but to avoid the discard of other changes, he needs to merge them with his modification (we suppose to have a versioning tool which prevents simultaneous updates by forcing the developer to update his work). Since neither of them knows what the other developer has modified, simultaneous changes may be in conflict. Thus, the developer who performs latest has to perform what is called a **merge**, which means resolving conflicts. This task may be very long and hard to carry out, so developers need tools to deal with it [6].

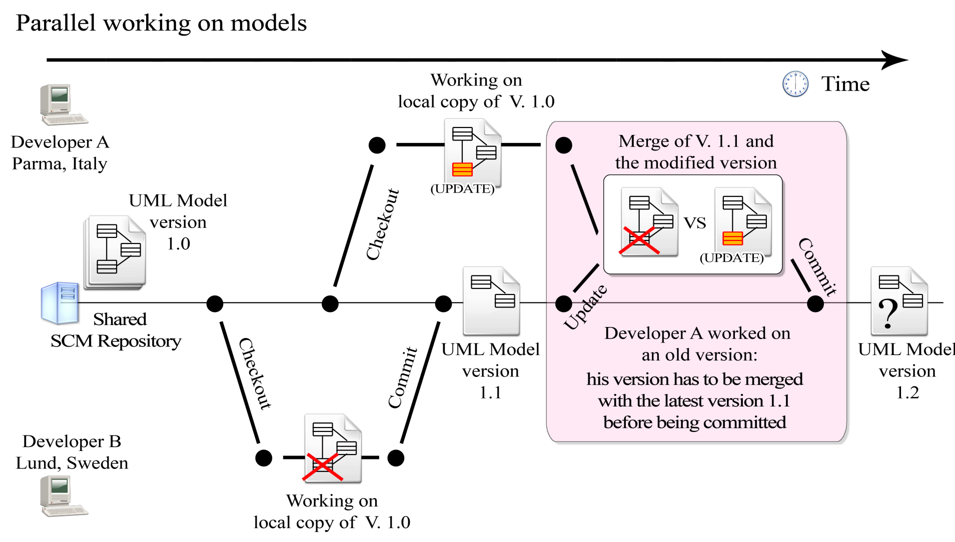


Figure 1: Merge of models

Moreover, if we blend together models and parallel working, we encounter the problem of performing a merge on artifacts which are models. The aim of our work is to recognize automatically conflicts and other problems caused by the simultaneous application of changes on two artifacts and to show them in a merged file. This should help developers to carry out the merge task.

Our approach does not use directly models as artifacts, as developers create them with an editor, which has its own way to represent models within the tool itself. However, all tools have to use a way to serialize models in order to save them. The serialization is performed using a standard markup language called XMI (XML Metadata Interchange) [14]. This is very important, because it implies that, theoretically, every model could be compared at the XMI level. This is the reason why we choose XMI, i.e. to be independent from the editor (as we will see in chapter 3, though, that this is not completely true).

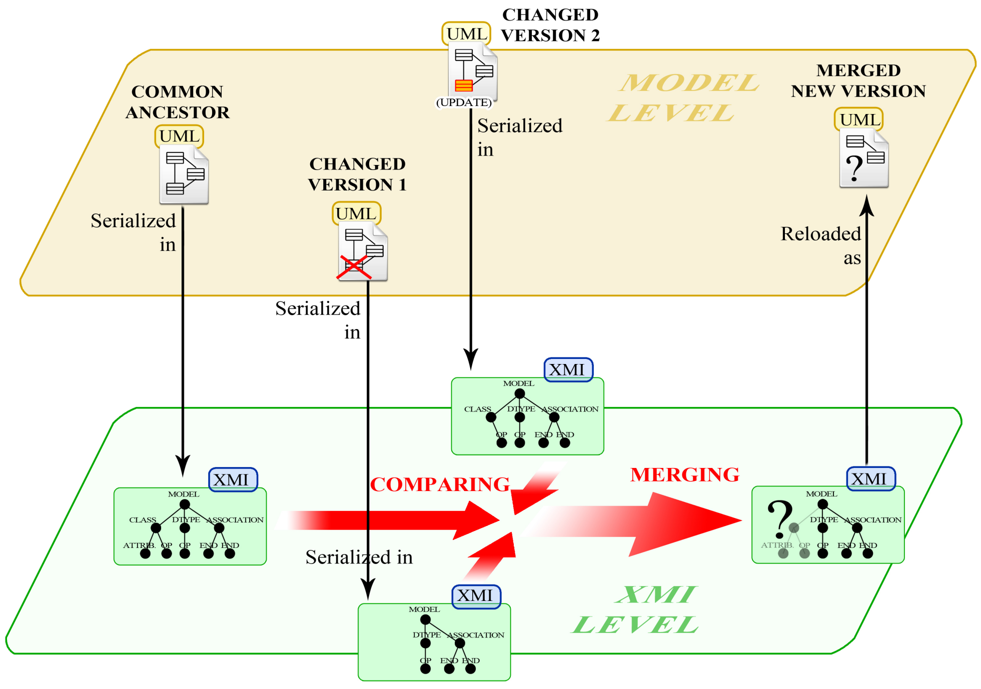


Figure 2: Models and XMI: we compare and merge at the XMI level

In fact, as we can see in Figure 2, every model is serialized as an XMI file and then reloaded by the editor. We work on the XMI area, comparing 3 XMI files which represent two simultaneous versions, which had changed the common ancestor, and the common ancestor itself. The result is a new XMI file which should represent a new UML model, the merged one.

The ultimate goal of this research is having a perfect merge on the model level. Our approach is far from performing such a merge, but it consists of the production of a merged XMI file obtained by looking solely at the information about the XMI syntax. This way, we remain independent from the model type (such as UML, Petri's Net, SysML, etc. and their versions) as well as from the editor. This means that we do not use any model semantic but only the one we can extract from the XMI structure.

What we have just described is called a state-based approach. Working with XMI, we could not consider the operation-based approach [11], since it relies on comparing two sequences of operation performed simultaneously: such an information should be extracted by consulting an editor which had recorded them.

Instead, as we have said before, we want to be independent from the editor.

Since we want to produce an XMI file as a result, we find it natural to work in a batch mode [8]. This is an approach which has not been tested yet, since most works dealing with model merging rely on the interaction with the developer, suggesting correct alternatives and providing (sometimes) a model valid merge. The problem with interactivity is that it makes the task of merging long and it creates the necessity of being entirely followed by the user. Furthermore, an interactive tool often forces the developer to choose his own order of analyzing conflicts, which means choosing the right alternative in that order, following the "path" selected by the tool. The problem in such an approach is that the developer cannot see the whole picture: sometimes the right decision should be taken evaluating a set of problems all together, because solving them one by one may result in not achieving the desired solution. In other words, the user should be free to choose his own way to analyze problems and then to find his own solution (that often is a completely new one, and not an alternative between the previous two). The interactive approach has often the side effect that the tool tries to provide solutions to all the conflicts or inconsistencies caused by simultaneous changes. Instead, we would like to create a merged file in which we apply those changes that do not cause inconsistencies (highlighting them), but we do not make decisions about those which do. The main goal is not to create automatically a perfect merge, since we think it is impossible or at least very hard, especially taking into consideration only XMI. Instead, we provide the user with all the information about changes, conflicts, inconsistencies and context-related problems which could be used to find the best solution by the developer himself (or by some other future tools which elaborate the given result).

2.2 XMI

The XML Metadata Interchange (XMI) is an OMG standard for exchanging metadata information via XML [14]. In other words, XMI is an XML dialect proposed to serialize models. Every model instance (for example a UML model) is derived from its metamodel (for example the UML metamodel). Moreover, we have another and more abstract metamodel called MOF that should describe the other model language metamodels (figure 3). To serialize a model is used the scheme in the figure 4.

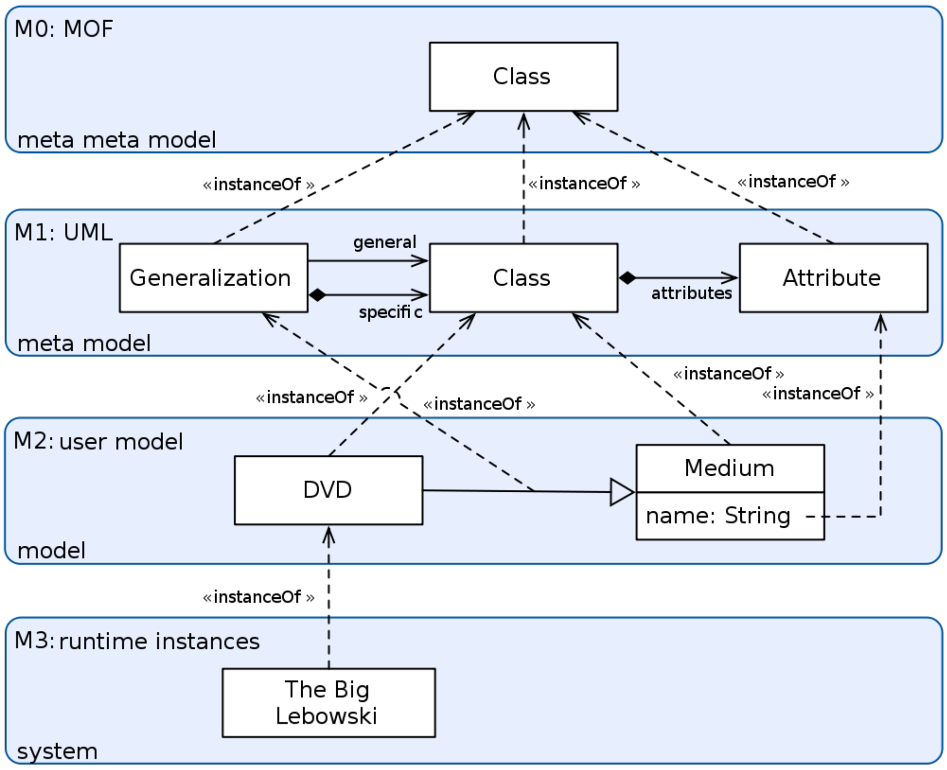


Figure 3: 4-layer metamodel hierarchy [18]

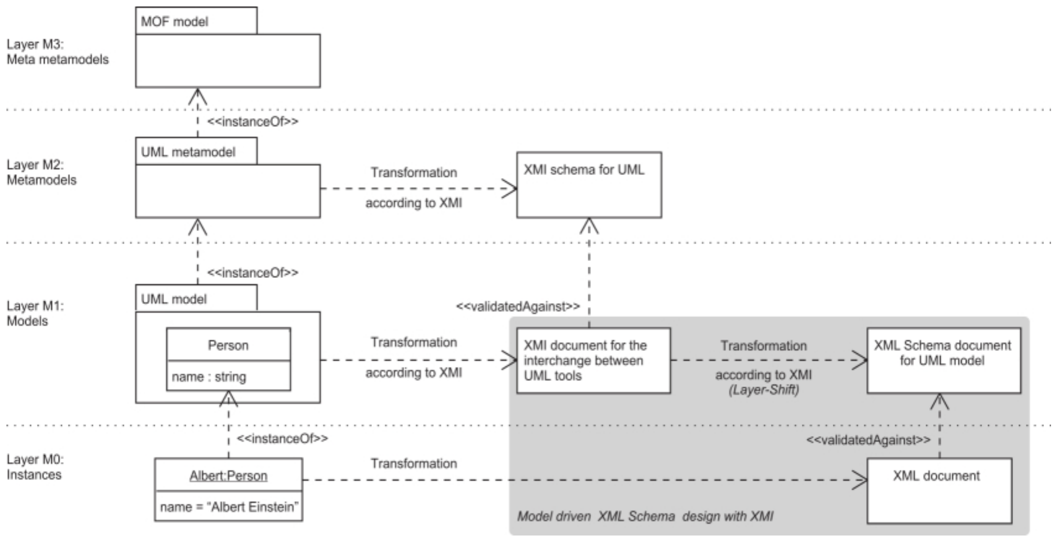


Figure 4: 4-layer metamodel and XMI [15]

Unfortunately, whilst here it seems to be a set of standards to define the serialization of models, in real implementations we do not have all this homogeneity. In fact, we have several versions of XMI, where the 2.x are radically different from the 1.x series. Moreover, we could have different files which are serialized using different patterns. These XMI problems, together with the fact that we have different versions of metamodels as well (for example we have several versions of the UML standard) and the different modeling tool vendor implementations, lead to a huge incompatibility between different XMI serialized models, as mentioned also in [16]. This means that we cannot just take three XMI files and compare them to obtain a result. This is a great obstacle to the realization of a useful merge result based on the XMI syntax. Therefore we are forced to take an XMI specification (the 2.0), choosing a pattern of serialization (even though we proposed a preliminary solution that covers all of them) and work on the assumptions we could extract from those. However, many considerations and assumptions we make here (with right adjustments), could be put in practice as well, once a stable standard will be provided. This is demonstrated by our implementation of a Java tool (XMIMerge). We have also some proposals for the extension of such a standard to support the important task of merging files.

2.3 Model serialization using XMI

Now we describe how XMI serializes a UML model, in order to make the rest of the work more comprehensible. Since every XML document is structured as a tree, the serialization patterns create an XMI tree in which elements are described by the model in the following picture (figure 5):

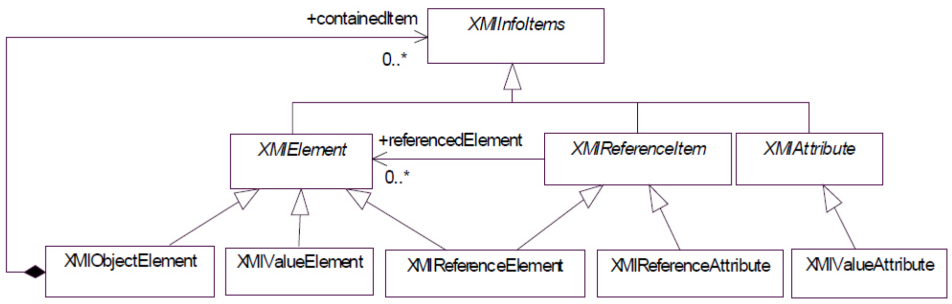


Figure 5: XMI description [14]

The root of the XMI file could have several child-nodes: we are interested principally in the *model* sub-tree. Everything about the logical part of the model is placed here, where other elements placed out of this sub-tree are concerning editor-oriented descriptions. We do not analyze them, though, since we do not know anything about the tool (including the layout, that we do not consider). For this reason, in the followings, when we mention root, we refer to the *model* node (the root of the model description) and not to the root of the whole file. Furthermore, there are some special tags of XMI such as *documentation* and *extension* which allow tools to put their own data about the model beside the logical model, without interfering with the meaning. This is a very useful feature of the XMI specification which allows us to consider only the logical model (in which we are interested) without having to find it, since it is kept separately and clean from other things. As we will see in section §3.2.5, the extension tag could be useful to implement a valuable feature for merge, that is the highlighting of annotations.

Given that every child-node of the root represents a class or an association in the MOF metamodel, we have the main problem of choosing the pattern of serialization. In the specification we can choose to represent every MOF class as a separate child-node of the root or we can nest classes as a child-node of the child-node and so on, representing their composition characteristics. This way, if we have a class `C'` which has a composition link with another class `C`, we will find `C'` as a child-node of `C`. Instead, with the former representation, every classifier is a different sub-tree of the root, and the composition link is represented by a reference (we will speak about these later in this section) or an association (which will be another sub-tree). The latter representation is more useful for our purposes, since we can state that every sub-structure of the root is a different MOF class or association, and, as we will see, we can gain advantage of this information to make helpful assumptions in the merge process. In fact, if we know that every sub-tree of the root is an entity, we can deduce that all entity moves are performed by references instead of moving sub-trees. This avoids a lot of possible move situations that we do not need to consider. For example, a refactoring will rarely change the tree structure, since the links between entities are expressed by references. Furthermore, we are sure that a node of a given level will not be moved to another level, since it represents a specific kind of feature. For example, a second level node will be an attribute or a method, but it could not be a multiplicity.

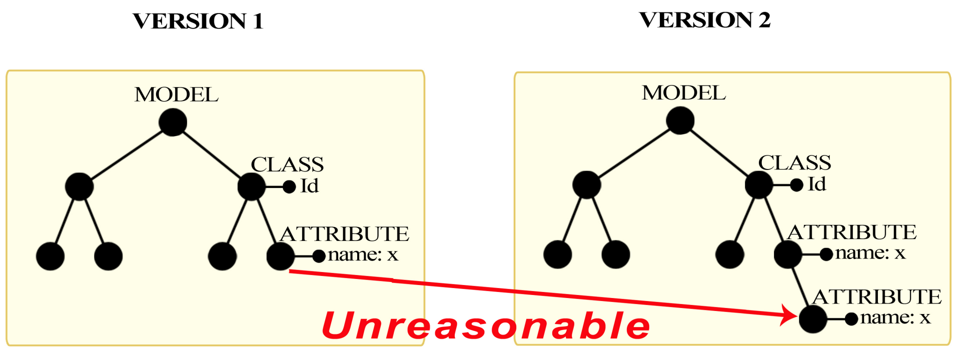


Figure 6: Moving an attribute to another level (for example beneath another attribute) is unreasonable

Besides, note that choosing one or the other way to serialize is equivalent, since it does not change the meaning of the model. Moreover, we use some examples which were created following this pattern. For these reasons, we will work principally using this pattern, then we will find a general solution that could involve also the other way of serializing MOF class. Our tool (XMIMerge)

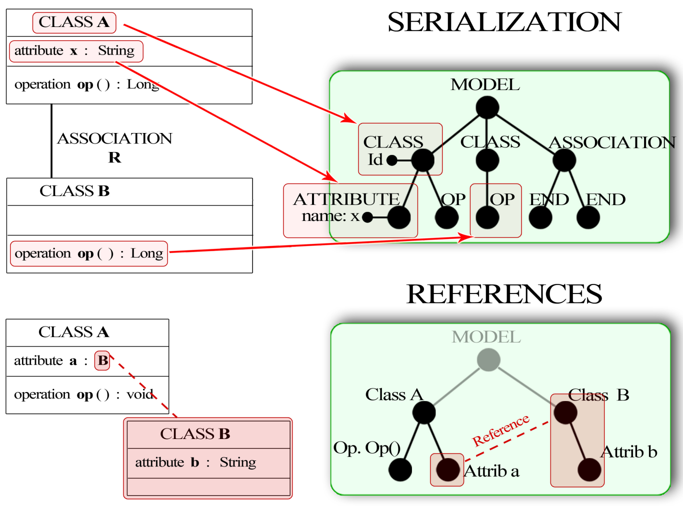


Figure 7: Serialization correspondences

handles artifacts serialized by ArgoUML, which uses this pattern, consequently, we had to deal with such a pattern.

As showed in figure 7, every class is represented as a first-level node, e.g. we can call it `F`. Then, every feature `F'` of such a class, like attributes or methods, are represented as children of `F.` Again, every feature of these `F'`, like the parameters of a method, is nested into the node `F'` as its sub-tree and so on. For a class diagram, which is the type of diagram that we studied in most examples, the average depth is five levels (it also depends on the tool). Features that are not meant to be nested, like the name of a class, are represented as XML properties of the node (that could also be a node without Id). Since we could create a misunderstanding, we decided to speak about *properties* regarding XML (and so XMI), while we call *attributes* the attributes of a class in the domain of models. Every property belongs to its node and it is not related with any other node (in XMI syntax), since it describes a characteristic of that specific node: this assumption will be used later to state the independence between properties (apart from references, that are described in the followings). It is important also to mention that we consider XMI values as properties: in fact, having a property of the form `p=v` where `p` is the name and `v` is the value, or having a node (without an Id) tagged `n` containing the content `c`, means the same thing to us: `p` is the same of `n` (the name to recognize the property) and `v` is the same of `c` (the value of the property). The only difference is in their format: an XML property cannot be very long and structured, whereas the content of a node could (since XML is a markup language, the node content is represented as everything between the start tag and the end tag: it could be almost everything, whereas the value of a property is just a string and has a restricted format). However, for our purposes, we consider them like two properties `p` and `p'` with the values `v` and `v'`.

A very effective mechanism provided by XMI is the property **Id** (which enable us to create global and local identifiers). With the Id every node can have a property that uniquely identifies them, and they are reachable without relying on their path from the root (as we will see in the merge process, this mechanism is very important). The Id is very useful also for matching the trees we want to compare. At this point we are faced with a new problem concerning the serialization patterns, as the Id is strongly recommended but it is not compulsory (you can save your XMI without Ids). Being that this is not so frequent in practice, we will work with Ids. Moreover, we have seen that some tools (like Rational Tau) had not kept the same Id in the same node in two different versions (see matching problems in §3.1.1).

Furthermore, there is another type of property defined by the XMI specification: the **reference**. This property contains as value the Id of another node in the document or in another document. (However, we assume that we have the whole model in one file: it does not change anything, since we can easily parse the two files separately and build the entire XMI tree.)This is a way to represent a concept like the *type* one: as we can see in figure 7, if we have an attribute a of the type `T` (that must be another `Class` or `Datatype` of the model and so another MOF entity and consequently a sub-tree of the root), inside the node which represents such an attribute we will have a property whose value is the Id of the Class named `T`. A reference is also used in the `Association Ends` of an `Association` to link the two classes involved in the described relationship.

Chapter 3

XMI merge process

In this section we present the merge process. We state some requirements and desired features that we would like to be satisfied in a batch model merge, and we will see which one of them is possible to satisfy (entirely or partially) using XMI. We show how such a process could be divided into five logical parts which could be studied and implemented separately. These parts are: change detection, conflict detection, interpretation, merge rules definition and changes application.

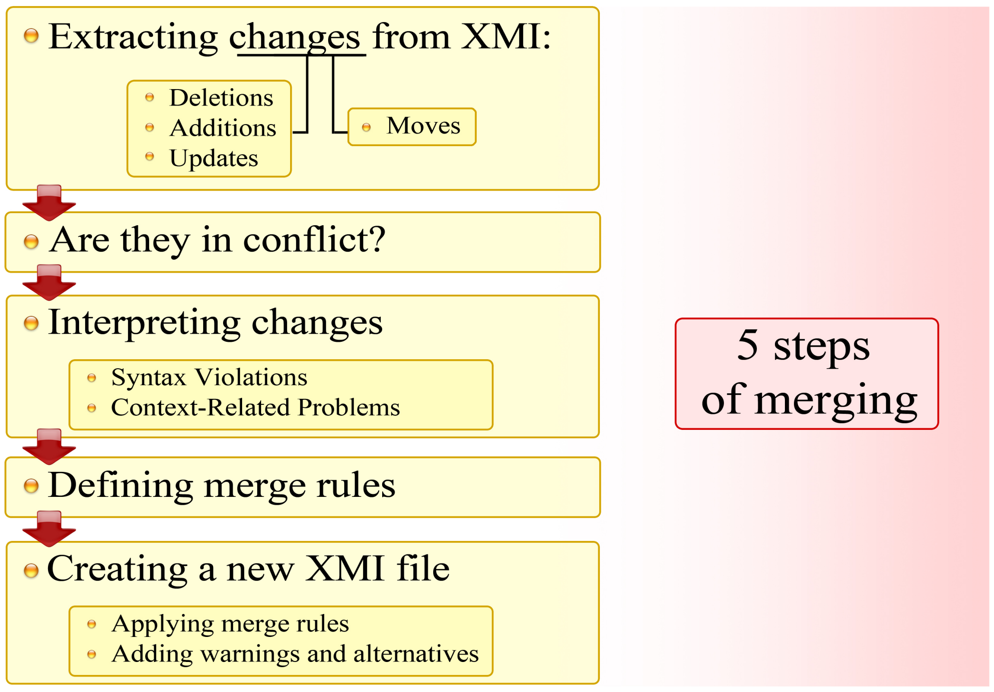


Figure 8: 5-step merge

3.1 Requirements (analysis)

In order to produce a correct merge, there are some requirements that have to be satisfied. We need three matched XMI trees, then we have to find changes and conflicts among them. The most important requirement in the result is the complete lack of loss of data.

3.1.1 Match

First of all, we have 3 files and we have to compare them. This means that we have to **match** the same elements in all files, in order to find changes among the 3 different versions. Every element needs to be recognized by an identifier, and XMI provides a mechanism to handle this (see §2.2).

There are two ways to use identifiers for matching. If the Id of a given element is kept identical when a new version is saved by an editor, we will have the same Id for the same element in all versions, thus we already have a match. Otherwise, we need an algorithm which recognizes similarities among the elements of the different versions and which reports the case in which we deal with the same element of two different versions (on the base of some sort of mechanism). At this point we are faced with a problem: existing mechanisms often recognize elements using their similarity, so it is less probable to recognize an element with a considerable amount of changes. In a merge process, we also have to detect all changes. These two concepts lead us to the conclusion that the more changes are present in an element, the less probable it is that a similarity-based mechanism recognizes the element, thus the more information we lose for the merge. To conclude, it is useful not to match with a mechanism that uses similarity for comparison, if we want to use the match result to find changes (like in a merge algorithm). There are several works dealing with such a match algorithm, both tree and graph based ones, and we can also find two works about specific XMI matching in the related literature [4]. Matching is often a very_expensive computational task, and it often causes the failure of merge algorithms [12]. We decided to concentrate our efforts on other issues and challenges concerning the XMI merge, for two main reasons: the matching problem is a well explored field of research, its analysis could be very expansive (could in itself be the topic for another thesis) and there is a simpler way to handle the problem concerning the XMI context. Thus, for the following part, we assume we have a set of already matched files (by their Id).

As explained in section 2.3, a generic node is composed by its properties (XML attributes) and its child-nodes, which are again composed by properties and sub-nodes, and so on. The leaf-nodes, however, are composed only by properties. We rely on the fact (in accordance with XMI) that we have a unique identifier for each node of the model tree. This means that we reach a node simply through its Id and it is independent from its structure position (like the path from the root). Inside every node, we have a set of properties which have a unique name, valid only within the scope of the node they belong to. This means that in order to reach them we need to access the node before, i.e they are node-dependent: to reach them we should refer to the node name plus their literal name. For example if we have a property `p` which belongs to the node `X` we will refer to it as `X.p`.

3.1.2 Change-detection mechanism

Once we have three matched XMI trees, we can look for changes among them. Since we know that both modified versions `V1` and `V2` are derived from the common ancestor `CA`, we only need to compare each version `Vx` with `CA` in order to understand which changes were performed to obtain `Vx`. This way, we would be able to apply all the changes on `CA` to create a merged file containing all the changes.

We have to choose a way to **localize changes**, and we would like it to be as fine as possible, in order to recognize as many independent changes as possible. For example, if we choose classes to localize changes, every change concerning that class will be represented as “class `C` changed”. Thus, suppose that a developer `d` has changed the class `C` modifying the name of a method `m`, and a developer `d'` has changed the same class modifying the name of an attribute `a`. Clearly, the changes are not related (or maybe they are in a more semantic way, but it could be analyzed later), that is, they do not affect each other. However, considering both of them as “class C changed”, we deal with exactly the same change affecting the same element, the class C, while we would prefer to consider the method and the attribute as different elements. In other words, we need a **unit of comparison** [13].

Furthermore, we need a mechanism to **describe changes** in order to analyze, compare and apply them. In the example above concerning class C, we do not explain exactly what we have to apply in the merged file. We should be able to recognize the nature of the change (e.g. deletion, addition, update, move), the part which has been modified (e.g. the name of the class, the value of the attribute, etc.) and how (e.g. the name of the class is now “D”, the value of an attribute is now 3 instead of 5, etc.).

3.1.3 Conflict-, violation-, and probably connected change-detection mechanism

When changes are detected, we need to compare them. We could come across changes that are **incompatible**, because they cannot be represented in the same file. In this case we are faced with a **conflict.** For instance, suppose a developer d modifies an element E and another developer d' deletes it: how can we represent an element which is updated and deleted at the same time? Obviously, if we apply the deletion, we will not see any update on it, since we cannot see it at all. On the contrary, if we can see the update, clearly we can see the element E, so we lose information about its deletion. Another example could be changing simultaneously the value of the same property p from 2 to 3 and from 2 to 4. How can we represent p that has both the value 3 and 4? We cannot. In the examples above the conflict derives from the fact that the same unit of comparison has been changed. Thus, we cannot apply both changes at the same time. Nevertheless, we have to highlight them in order to make sure that developers can manage it. A very important requirement to find all conflicts, is to define carefully a unit of comparison. The more coarse it is, the more false positive conflicts we find.

Furthermore, we could discover that two changes (placed in different changed versions), when represented together in the merged file, would **break the validity** of the XMI syntax, while separately they did not. We speak about **violations**: we should detect these changes and we should report them. An example could be a reference pointing to a deleted element, i.e. the value of the reference in `V1` is the Id of a node that doesn't exist anymore in `V2`. This can be represented clearly in the final file, but it brakes the XMI syntax.

Finally, there are changes which are not directly related and which together are not breaking the XMI syntax. However, changes could still be close to each other. Probably (even though we cannot say exactly) they are **related** when we consider the model-metamodel (for example UML or any higher constraints system like OCL). It would be useful if the user were warned about a probable relationship (at a higher level) between two changes. We can explain this better with an example: a reference has been changed to point to the node `N` in `V1` and an attribute of `N` has been modified in `V2`. The developer `d` working on `V1` is linking an element to another one which in the meantime has been modified without `d`'s knowledge.

3.1.4 Avoiding loss of data

We should make sure that all information about every modification (of both versions) with respect to the common ancestor is present in the merged file. We can represent the information about mergable changes simply by **applying** them. Moreover, it would be useful to highlight where exactly the changes are applied to enable the developer to localize them easily and to verify them.

In the case of those changes which could be merged but which would violate the syntax (like XMI syntax), we have to make a decision. We can either choose simply to apply such changes which, however, would result in an invalid XMI file or we can **discard** one or both of them to obtain a valid XMI file. In the latter case we have to report all the information regarding the changes and also not having performed it (them).

As for those changes which could be related on a higher level (that we have no or little knowledge about), we should apply them and insert a **warning** about the fact that they could be related.

In the case of conflicts we cannot apply the related changes because they are not simultaneously representable, so we need some sort of mechanism to represent both **alternatives**. To do that, we have two options. We can apply one of the changes (but which one?) and create a mechanism to represent only the one we have not applied as an alternative to the change we performed, or we can represent both alternatives with the mechanism used in the case of non performed changes. In the latter case, we can decide to leave the original solution of the `CA` and then connect the alternatives to it, or we can omit the whole interested element.

3.1.5 Symmetry

Even if we perform the same merge several times, we should always obtain the same result, that is to say, the outcome should not depend either on the order of detection, on the management of changes or the different order of the input versions. For example in the Lindholm merge [12], a conflict is resolved by choosing the solution proposed on the first loaded version. This means that if we have loaded the version `V1` and `V2` we will have, in the merged file, the `V1`'s solution of the conflict. Contrarily, if we have loaded the version `V2` before, we find its solution in the merged file. We would like to avoid this kind of results.

3.2 Merge process

In the followings we describe a possible way to perform a merge process based on XMI which satisfies the previously mentioned requirements. We divide the merging process in five major logical steps, which could be studied and implemented as distinguished sub-topics. We begin with **detecting changes** (step 1), then we compare them to each other to find unsolvable **conflicts** (step 2), then we **interpret** them to recognize violations and hypothetical context-related problems (step 3). The fourth step (4) consists of defining a set of **merge rules** to handle the previous problems and finally, in step 5, we create the **merged XMI** file.

As we will see in the part concerning the explanation of the algorithm and the implementation of our tool (XMIMerge), the order is important because one step requires an output from the previous step. However, sometimes it could be convenient, at a practical level, to anticipate the task of a step as soon as we have a partial output from the previous step.

3.2.1 Change detection

First of all, we have to identify the changes between the common ancestor and the changed versions. As discussed in the paragraphs about requirements, we need to **localize and explain changes**.

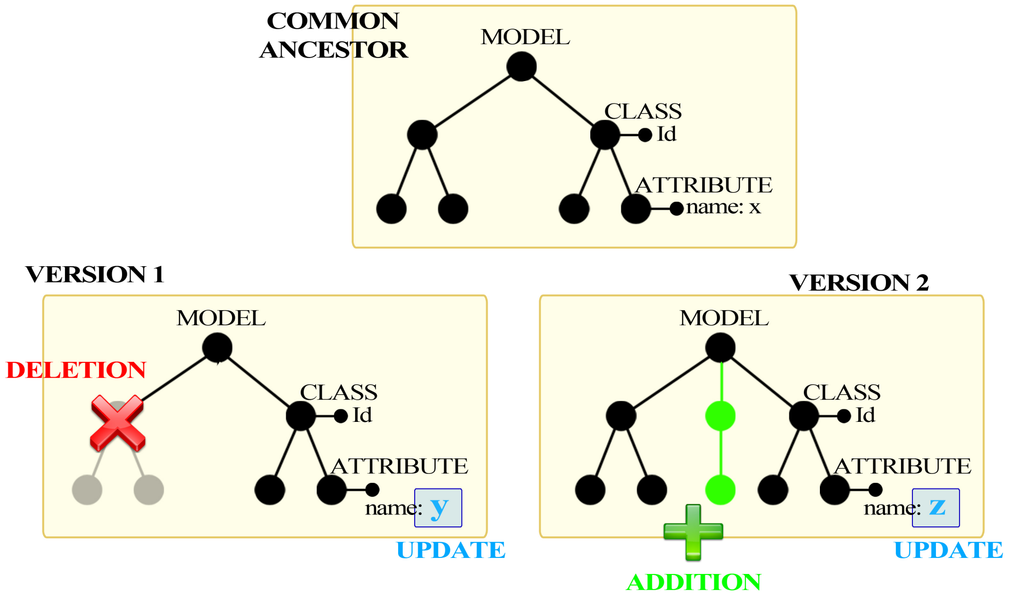


Figure 9: Some examples of changes

As explained in §2.3, a generic node is composed by its properties (XML attributes) and its child-nodes, which are, again, composed by properties and sub-nodes, and so on. Every node contains many sub-nodes and properties, whilst the leaf-nodes are composed only by properties. We can state that **a node is changed** if and only if one of its contents is changed: a sub-node or a property can be added, deleted or updated (as in figure 9). Moreover, we consider a property (or rather its value) as the smallest atomic element that could be modified, as we cannot split its value in more parts. Later, in the part presenting conflict detection, we will discuss a particular case in which we prefer to relax this constraint.

We can, moreover, encounter the case in which a node X is changed because one of its property `p` or one of its sub-node `Y` has been changed. We could represent a property change writing `[X, up(p, op())]` which means that the property `p` of the node `X` is changed: `op` could mean `del` for deletion, `add` for addition and `up` for update. For updates, we would like to specify some further details, the reasons for this will appear clearer in the next section about conflicts §3.2.2. Thus, we could write `[X,up(p, up(v'))]`, which means that `p` is updated with the value `v'.` Furthermore, a property could also be a reference (see §2.3). In this case, being that it is used as a mechanism to point from one node to another, we can describe the change of the value of a reference property `r` as `[X, up(r up(Y→Z))]`, which means that `r` is now pointing to `Z` instead of `Y.`

If a sub-node is changed, we can write the propositions `[X, up(Y, del)]` or `[X, up(Y, add)]` respectively if we are updating the node `X` by deleting or adding a node `Y` from (to) the node `X.` Both these propositions mean that all their sub-elements are deleted or added. Consequently, we call them **composite changes**. If a sub-node `Y` is modified, we should describe its modification further, exactly as we did in the case of the parent node `X`. This means that we could have the sequence `[X, up(Y, up(...))]` until we reach a leaf, where occurs a property change. In other words, this statement represents the path from the root to the changed property.

At this point, we introduce the **move** change. Even though we can consider the move change as two different operations (deleting a content from a node and adding the same content to another), the representation of move will be useful later, when dealing with change interpretation and conflict detection. First of all, we cannot recognize a property move, since a property is node-related (as explained before, we could find two properties with the same name in two different node, and they are identified by the Id of the node to which they belong plus their name). In fact, a property only describes the node to which it belongs, and moving it means simply creating a new one. Differently, nodes could be moved from a parent-node to another. In this case, we do not have to replace the existing representation (`add + del`) but we can represent the change as a new proposition: `[Y, move (X,Z)]` which means that a sub-node `Y` is moved from the parent-node `X` to the parent-node `Z`. Obviously, a move is a composite change. The move can cause a problem with the approach of change detection mentioned. In fact, we identify a moved node `Y` by its path from the root: moving it means that the path has changed. However, we should recognize changes in the node `Y` even if it is the child-node of `X` in the version `V1` and that of `Z` in the version `V2`. In fact, it is still the node `Y` with the same identifier, so it will be the same MOF object. To do this, when we find a move, we should repeat the change detection on the moved sub-tree, considering `Y` as if it was not moved. This way we can flag changes with respect to the `CA` contained also in the moved elements. Let us consider, for example, the case in which the node `Y` has been moved from `X` to `Z` in the version `V1`, and the property `p` belongs to `Y`. Suppose that the developer `d` has moved the node `Y` and has changed the property `p`. Before recognizing the move, the node `Y` has not been compared with its namesake in the `CA`, so we do not know that `p` has been changed (in fact we have recorded only that a node `Y` has been deleted and another node `Y` has been added). Once we know that `Y` has been moved, we can compare the moved `Y` (in the version `V1`) with its namesake in the `CA`, because now we know that it is the same node. Thus, we know that it has been changed because its property `p` has been modified. How should we record this change? We should use the path in the `CA`, and not the new one (the one in `V1`), because, in the conflict detection part (as we will see in §3.2.2), we will compare `p` with its namesake in `V2` and a change will occur on the same element if its path is the same. For this reason, we write the change as `[…X, up(Y, up(p, op(...)))]` with `X` instead of `Z` (the node under which `Y` has been moved).

Finally, since we have to specify in which modified file we found a change, we should include in the above statements also the version: the statement will be of the form `Vx[stm]`, where `Vx` is `V1` or `V2` and `stm` has the form described above. We can refer to the changes by assigning them an arbitrary (but univocal) Id.

Note that this representation of changes avoids infinite propositions, since it is constructed over the MOF tree structure (see §2.3) provided by XMI, and not over a graph. This representation has more benefits which will be described later on.

3.2.2 Conflict detection

Our next step is finding **conflicts** among changes. As we said in the requirements part (§3.1.2 and §3.1.3), we need to find a *unit of comparison* (we will call it UC) which should be as fine as possible. We could take the *property* as UC. In fact, we stated before that we consider it (or rather its value) the smallest atomic element that could be modified. We are faced with a conflict when the value of the same property is changed (see Figure 10 where the value of the property `name` has been changed in `y` and `x` simultaneously), since we cannot represent two values simultaneously. Furthermore, if we delete a property and we update it simultaneously, we create a conflict too, because we cannot represent the updated value and the "lack of property" at the same time. If we add property p to the node `X` in the version `V`, it could create a conflict only if in `V'` we add or modify the same property `p` (inside the same node `X`). In this case, there is a conflict, exactly like in that in which we updated the same property. Otherwise we have no conflict between the properties. In all these cases, we detect a conflict on the property `p` of a node `X`, that will be managed later, and we can mark it.

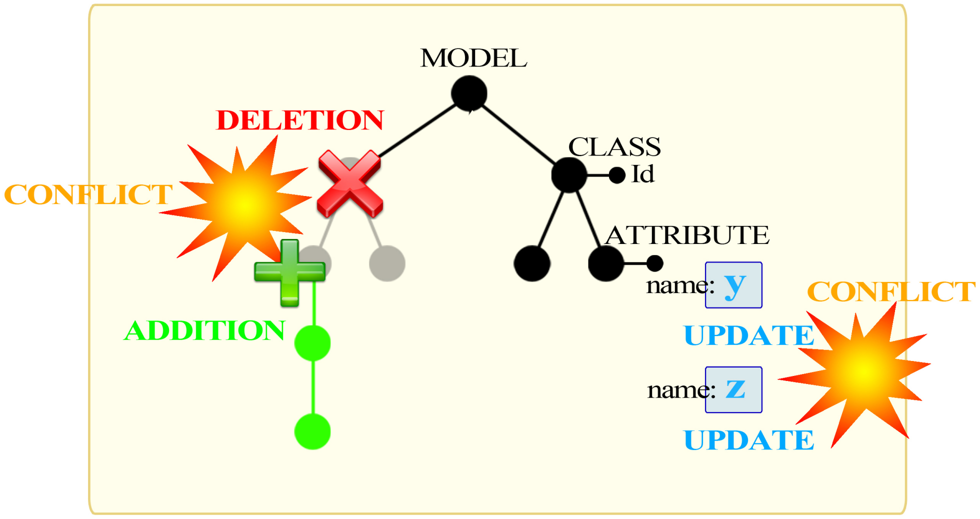


Figure 10: Some examples of conflicts: the addition of a node as a child of another deleted node and the update of the same property.

But what happens if someone deletes a node? In that case how can we use the property as unit of comparison to find a conflict? Having the UC as property, we could add a change for every property that belongs to the deleted node, and say that it was deleted: note that, this way, we split a change like `[X, del]` in many changes like `[X, up(p, del)]` for those properties which belong to `X`, while as for those that belong to a sub-node `Y` there are many changes of the type `[X, up(Y, up(p, del))]`, and so on, for every sub-node. Since we could have many properties, this way we would create an explosion of changes. Moreover, and more importantly, we lose the information about the fact that the original change was on `X` and not on a sub-part of it. In fact, the whole node `X` was deleted, which means that, due to the hierarchical structure of XMI (§2.3), we should consider the deletion of an element with all its description, and not as a collection of many changes. Thus we should link the two changes as conflicting ones: deleting a node `X` and updating one of its sub-parts at the same time. Clearly, the situation is even worse when a node `X` is deleted and a new node `Y` is added under it (Figure 10).

In our solution we prefer a **dynamic UC** to find conflicts, rather than a fixed one. Since a model in XMI has a hierarchic structure, we can use an approach like the one explained in Asklund [2]. We can compare the parent node and, if there is a conflict, we can go deeper until we find it on the smallest node content. Consider a node `X` as the root of a sub-tree of the model node (which we do not take in consideration). If we have two simultaneous modifications within it, we will have two changes of the type `Vv[X, op(...)]` and `Vv'[X, op(...)]` where `op` could be any possible change, `v` and `v'` are the changed versions (the order is irrelevant). If we do not find such changes beginning with the same node `X` as prefix, we can claim that there are no conflicts inside the node `X` and all its sub-nodes, due to the construction of the changes (for the moment we ignore moves). However, if we have such changes, we can go a step deeper examining them. With one step, we mean that we consider the next sub-node on the node-path described in the change. At a certain point, we could find that the two changes could be exactly the same to the end: in this case there is no conflict, the two changes are equivalent. Otherwise, we could have several type of differences:

- two different nodes were modified, so there are two changes of the form
  - `Vv[...X, up(Y, op(...))]` and
  - `Vv'[...X, up(Z, op(...))].`

  In this case we know that whatever the changes are, they involve two different nodes (and then two different sub-trees, see Figure 10), so we can deduce that there will no be any conflict between these two changes. In fact, they are not placed in the same part of the tree, and they cannot involve the same element.

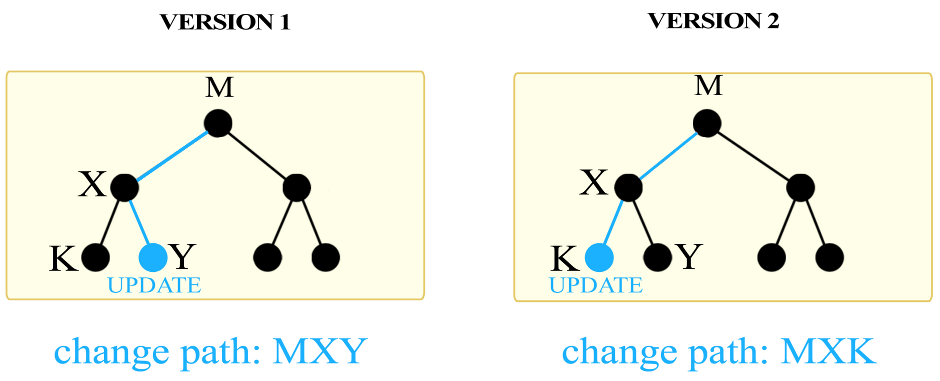


Figure 11: Two updates have different paths

- the same node is changed. Then we can encounter the following cases:
  - it has been changed with the same operation in both versions. If it is a deletion, we have reached the end of the proposition and the two changes are identical. (As stated before, there is no conflict because the changes are equivalent). There cannot be two additions of the same node (as assumed in §2.2). Furthermore, there remains the case in which the same node is updated: we have to go ahead with yet another step deeper.
  - It was changed with two different operations which could be only a deletion and an update. In fact, there cannot be a node which was both added and changed with another operation, because adding it in the version `Vv`, means that it did not exist in the `CA`, so it could not be

modified in any way in the version `Vv'`. Thus, in the remaining case, in which the same node has been both deleted and updated, we certainly have a conflict. In fact, the deletion of the node `X` in `Vv` is conflicting with any other change that could be represented by an update change of `X` in `Vv'`. We do not need to check deeper, we know that there is a conflict between these two changes and we have to manage it.

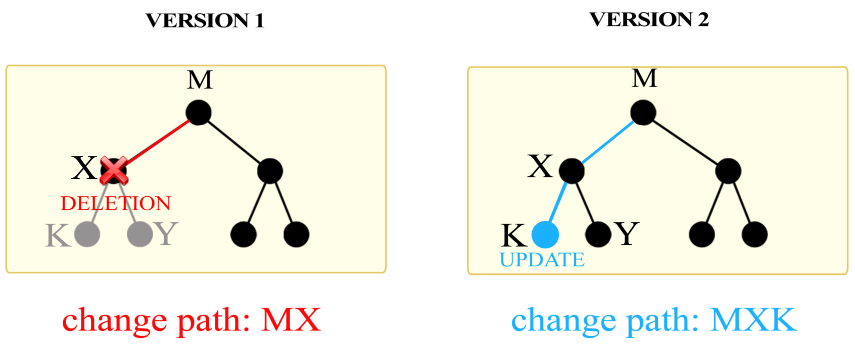


Figure 12: The two changes have a common prefix in the path: since one of them is a deletion, there is a conflict

- A property is changed, as well as another sub-node. We have the same situation in which two different sub-nodes are modified. Two different parts of the tree are modified, thus there is no conflict.
- Two different properties are changed. Like in the previous case, there are no conflicts.
- The same property is changed. As described at the beginning of this chapter, in this case we always have a conflict.

To sum up, we have analyzed every pair of changes, and we have decided whether they were conflicting or not. The method is complete, since the above explanation itself contains the proof that it covers every case of a hypothetical conflict (without considering moves). More detailed explanation can be found in the chapter Discussion.

This way of detecting conflicts, allows us also to add an interesting feature. In fact, as it is based on the depth of the structure and not on a fixed UC, once we find an update/update conflict on the atomic value of the property, we can apply the idea of "going deeper". We can do this by running another specific type of algorithm on the value, which could be a long text or structured in a way different from XMI. In fact, considering the XMI/model field, it is possible (§2.2) for a property to be an XML tree itself, or a slice of code. It could be extremely useful for both of them to delegate the task of finding more conflicts to another and more appropriate algorithm (already existent), since the detection of these new conflicts brings us two advantages. If real conflicts emerge, we provide the user with a more detailed merge, whereas if no conflicts occur, we have eliminated a false positive. How to integrate such an algorithm is not discussed in this thesis.

We asserted that the conflict detection described is complete. However, we did not consider the **move** changes. We left it as last issue, because it is not a change like the others. Firstly, it consists of two different changes already analyzed, and secondly, as said at the beginning of this chapter, using a serialization pattern instead of another could avoid moves.

However, considering moves, a conflict can occur in two cases: if the same node is moved in different places simultaneously, or in the case of two particular nested moves. We can represent the latter problem in the following way: a node `X` is moved under (in the sub-tree of) the node `Y` in `Vv,` and another developer moves the node `Y` under the node `X` in `Vv'`. Clearly, there is a conflict, since we cannot simultaneously represent the node `X` as both the progenitor and the descendent of `Y`. However, the move cannot raise conflicts with other changes, as it is independent from them. In fact, moves can only involve nodes, not properties (because they are node-related). Thus, applying the move before or after another change, does not change the result. In other words, changes as deletions, additions and updates, modify the information contained in the elements, while a move simply changes their place. For a more detailed explanation, we can analyze these cases:

- move/add: a node has been added (`V1`) in a sub-tree which has been moved (`V2`). Adding a node before or after the movement of a higher-level node does not change the result.
- move/up: a move cannot be performed over a property. On the contrary,

updates always end in a property change. Again, modifying a property before or after a move yields the same result.

- move/del: the only problem could be if we delete the same node that is moved. Note that this is not causing a conflict, since the deletion of `X` is just a part of the entire complex change of moving `X`, that consist of deleting `X` and adding it again. If we do not delete the same `X` that we move, we could have two cases:
  - the deletion involves a higher-level node `Z`. We can apply both changes without conflicts, since `X` does not belong anymore to the sub-tree of `Z`.
  - the deletion involves a sub-node of `X`. In this case, again, we can apply both changes regardless of which one comes before.

In all these cases, we could insert a warning because we suspect that the two changes could create problems, but this is an interpretation issue (and will be discussed later).

We have proved that the move is independent from other changes, and it could not raise a conflict with them. However, considering the move change, we could be able to **avoid** some **false positives** and some **false negatives** reported by the previous conflict detector. In fact, consider a node `X`: if it has been *moved* (and not only deleted) in `Vv,` it results deleted with respect to the `CA.` Then, suppose that in `Vv' X` has been updated: there the detector would report a delete/update conflict. This is a false positive, because the node exists (it has just been moved) but its path has been changed and the previous detector fails to recognize it. Moreover, for the same reason, if some updates were performed in the sub-tree with root `Xv,` they will not be confronted with the same sub-tree with root `Xv'.` This means that we do not find the conflicts (because we do not compare them, as we consider them different nodes), so there are a set of possible false negatives.

These are problems (concerning the previous conflict detector) which derive from the use of the **path-strategy**, applied to find conflicts without take in consideration the *move* change. As explained before in this chapter (and in §2.3), moves are not so frequent or we can assume not having them at all, especially using a certain pattern of serialization. Thus, we could accept such an inconvenience (when it is really marginal), and we could decide to adopt the detection method described above.

However, considering the move changes, we can **modify the conflict detection** process by adding some control. In fact, we can simply ignore the conflict raised by a delete/update (where the delete is the non recognized move) on a node `X` because we know that `X` was moved and not deleted, avoiding this way to mark a nonexistent conflict. Furthermore, we can use the whole process of conflict detection described above to compare the sub-tree with root `Xv` (moved) with `Xv'` (updated). This is possible, because we have the same node Id thanks to which we can associate them. Conflicts are discovered this way, even if the path is not identical. Eventually, we can handle moves and discover conflicts. However, there are some problems of interpreting and representing moves and conflicts, which will be discussed in subsections §3.2.3 and §3.2.5.

3.2.3 Change interpretation.

Once we have detected *conflicts*, we have to check the changes for further potential problems among them, like **violations** of the XMI syntax or probable **context** issues. We deal with them together because they both require further information at a higher level (like considering the metamodel, running a validator or deducing some complex operations), in other words, we have to **interpret** them. In this subsection we explain the method that we used. As we discussed at the beginning of this chapter (and as we will observe in the algorithm explanation), the order of these steps is not strictly decisive, which means that it could be preferable, sometimes, to perform merge rules before the interpretation. Thus, sometimes it might seem reasonable to refer to a merge rule that could be already performed or we know for sure that it will be. Finally, this part is not strictly required for a batch merge [8], but it could be considered, studied and implemented as an independent task, to be carried out after a batch merge (whose result is a merged file that may not be XMI valid and model-semantic valid).

The interpretation part of identifying XMI syntax violations could be performed using an XMI validator on the entire file, once it has been merged. We preferred to perform such a job taking previously detected changes as inputs, and analyzing them to discover only the violations that they could provoke. Furthermore, as explained in §2.2 and §2.3, we might use different XMI versions and serialization patterns, so it could be hard (or even impossible) to perform a validation that covers every possible output file. Thus, we worked on the serialization patterns described in [10] and in the specification of XMI 2.0 [14].

The context-related interpretation cannot be precise because we have only a small amount of information deduced by the MOF structure and serialization pattern about such a context (§2.3). That is merely enough to warn about hypothetical problems. Moreover, finding relationships between changes is still an open issue in research which could be very complex to explore, as mentioned also in [8].

In the assumptions we set out the requirement for a series of valid XMI as input. Thus, we know that a change itself cannot cause a violation, otherwise the changed version would be invalid as well, and that is not possible. Therefore, we have to explore those cases in which a set of simultaneous changes *together* may break the validity of XMI syntax. In summary, we claim that a **violation** occurs when a **change affects another change** indirectly, i.e. it breaks the validity of the result of the other change (or vice versa).

In §3.2.2 we stated that a property change is independent from another property change. This is true and it holds in the XMI syntax as long as we don't consider references. In fact, a reference `r` is a property whose value is the Id of another XMI node `X`: in other words, `r` points to `X` (§2.2). This may be source of a set of hypothetical violations: in fact, in a valid XMI file, we cannot have a reference pointing to a node which does not exist. For this reason, every time a node is deleted and a reference to it is updated/added, a violation emerges (Figure 13). We can choose between two options to handle this situation. We can either leave the violation intact (keeping an invalid XMI) and warn the user about the problem (which could be detected later with a validator), or we can discard the deletion, reporting, in some way, (see the 5th step, creating merged XMI) our decision and the cause for it (to highlight a violation of the XMI syntax).

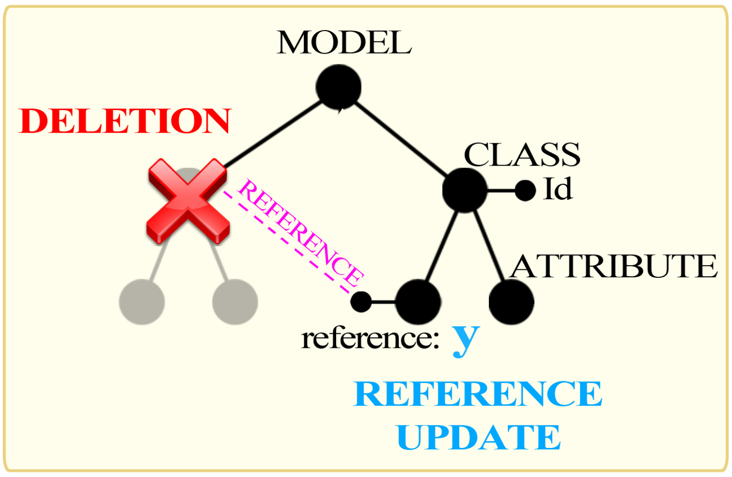


Figure 13: Example of violation: a developer changed a reference to a node previously deleted by another developer.

However, such a violation (from another point of view) could be also considered a probably related change, since in the version `Vv` the developer `d` changed an object `O` that now points to some other object `O'` (often in the class diagram a reference represents a `type` link, as explained in §2.3), while in `Vv'` the developer `d'` has deleted `O'` without being aware that an object `O` was changed simultaneously to point to `O'`. This means that probably these changes will create a problem also in the domain of models.

The difference between the two considerations above (considering an XMI syntax violation or a model-domain issue) is that in the first case we know XMI syntax and we can make a decision on the basis of a precise information; contrariwise, in the second case we are simply making suppositions about the model (context) related problem. In fact, the latter one is probable but not sure, and it could depend, for example, on the type of diagram used at a higher level (UML, etc.). This difference leads us to call **violation** the former problem and (hypothetical) **context-related** the latter one. However, in this case the second consideration *confirms* that there is not only a violation of XMI but also a probable higher level conflict. These considerations will be used to define a set of merge rules which handle the problems encountered. In this case, as we will see in the next section on merge rules (§3.2.4), we opt for the solution of discarding the deletion and report a warning, in order to maintain the semantic valid and to warn about a very probable developer-intention breaking. Discarding a deletion does not cause any loss of data, and a warning could be created with a very simple message.

Let us recall that a **violation** occurs when a change affects another change indirectly, i.e. breaking the XMI validity of the result together with the other change. Since we claimed that properties cannot affect XMI validity of other properties or nodes (except for the references, which we discussed before), we should consider only the composite changes (which involve more than one node). In fact, a composite change may affect other changes by modifying a node `X` that "includes" them, in the sense that the other changes are modifying a content which belongs to the sub-tree with root `X`.

At first sight, this seems hard to handle, since there might be nested changes which are related. Furthermore, applying a merge rule for one of them before the other, could modify the result, breaking one of the requirements (symmetry). However, a composite change always involves a node, and it can be of the following type: deletion, addition or move. We do not consider the node update a composite change, since, as we mentioned in the conflict detection paragraph, it always leads to another change, which could be one of those just mentioned, or a property change.

Thus, we have the following cases, in which:

- having a node deletion in a version `Vv` could not involve other nested changes: in `Vv` there are no changes involving a sub-tree of the deleted node (there are no sub-trees anymore), and every time a change is made in `Vv'`, it causes a conflict, provoking the discard of the deletion (as we will see in the merge rule part).
- when adding a node `X` on `Vv`, apart from the reference case already explained, there cannot be nested changes, since in `Vv'` we cannot have any change involving the sub-tree with root `X` (we do not have such a sub-tree at all, since it was not in the `CA`).
- only the move change remains, and, in fact, it is sort of a "Pandora's box". We could have many situations in which combining nested move changes

with other changes could cause a lot of violations and hypothetical context problems. Furthermore we mention again that there is a way to avoid moves (or at least strongly limiting their occurrence). However, we found some solutions to handle these problems.

The **first** and the **simplest solution** is to ignore the existence of such a change, seeing moves as deletions and additions (of the same node, with the same Id). In this case, we are faced with a problem when we have an update/delete conflict (see also conflict detection, §3.2.2). In fact, whenever a node `X` is *moved* in `Vv` it results deleted with respect to the `CA`, and if in `Vv'` `X` has been updated, a conflict occurs. The merge rule for this consist of discarding the deletion, and causing the duplication of `X`. This leads to an invalid XMI with two nodes with the same Id, and to have only one of them updated, while the moved node could not be updated since it is considered an added one from the change and conflict detector. The most important side effect of this approach is that if there are updates in `Xv`, they will not be confronted with the `Xv'`, which means that we do not find the conflicts (because we do not compare them, considering them different nodes). Thus, once a validator raises a problem about these two nodes showing that they are the same, the user is forced to check again them for changes and conflicts. Unfortunately, moving a big sub-tree means not finding a lot of hypothetical conflicts. However, in the pattern without nested MOF entities, where refactoring the model involves references (see §2.3 and second solution below), the hypothesis of not having moves is perfectly plausible. The following solution includes this one with the addition of a small set of reasonable and safe moves.

The **second solution** is connected to a specific serialization pattern used by XMI (the one without nested MOF entities). As we can see later and we have mentioned in the previous sections (change and conflict detection), this pattern has more characteristics which make our algorithm working better. Furthermore, it is equivalent to the other patterns, which means that using this one does not lead to losing information about the model, and another differently serialized file could be transformed in one like the this. In this pattern, every sub-tree of the root is a first level entity (a class or an association) in the MOF representation, which means that we cannot have a first level entity as a sub-tree of another entity, so we could not move an entire subtree. That also means that we have a short XMI tree (in analyzed class diagram the average maximum is 5, as said in §2.3) thus, there cannot be many nested move changes. Moreover, due to the hierarchical structure, every second level child-node represents a feature of the parent node; the same holds for the third level node with respect to its parent and so on. This means that the more deeply we observe a node, the smaller object it represents, the more parent related it is, thus, a move is highly improbable. In fact, since we have a lot of first level entities connected by references, their second level nodes represent attributes and method, and their third nodes are parameters of the methods etc. Clearly it does not make much sense to move a parameter from a method to another. It is easier for an editor to allow a user to write a new parameter inside a method specification: this means creating a new node with a new Id in the XMI tree. Finally, note that, with this pattern, the moves of classes in model domain, are performed in XMI changing references (we can handle reference changes without problems) and not the tree structure (for example a refactoring, see §2.3). This also means that there will not be many moves of nodes and that they do not involve entities.

For these reasons, we consider only non-nested *move* changes in this solution, and only *move* changes of a second level node. In such a case, we can have only a node moved to another substructure (sub-tree). How could it create violations? For the next cases we will not consider the option of leaving intact the violations on the merged file, unless we have to discuss some particular problem. Otherwise, leaving intact violations means exactly applying a change and creating a warning. Furthermore, whenever we are faced with a violation, it could obviously be a problem at a higher level: since the problem is highlighted already by finding a violation, we need to add nothing more to the user. Follow the violations caused by a move in a setting with the described constraints:

- move/move: the same node is moved in `Vv` and in `Vv'`. The violation consists in obtaining as result two nodes with the same Id in the merged file. A way to handle it is either to use alternatives or we can opt for discarding the changes and adding a warning about both moves.

- move/del: every time a deletion is combined with a move, not a violation occurs, but there is a context issue:
    - a deletion involves the parent node `Y` of the moved node `X`. No violations, since we can apply both changes without breaking the syntax. We could encounter a context issue, though: in fact, the deletion of `Y` could have meant the deletion of all its child-nodes, while the sub-tree with root `X` is present on the merged file (but it is moved). We should warn about the non-deletion of `X`;
    - a deletion of `X` and the move of `X` itself: the same statements explained before;
    - deletion of a sub-tree of `X` and the move of `X`. In this case, we suppose that moving `X` the developer does not want it to be affected by a deletion. Deleting, we lose information, so a solution could be to discard the deletion and to add a warning about what was "not deleted";
- move/up: there are no violations, since a property could be XMI-syntax related to the moved node, `X`, only by being a reference. In that case, the Id of the node remains the same, so if a reference was changed (added) to point to it, the pointer is valid also after the move. Of course, in this case, we have a probable context issue, because a developer is moving an element `E` and another developer decided to point at `E`. In this case we could insert a warning. Note that, as described before in §2.3, the pattern we use in this solution, combined with the class diagram, implies that we can have only a reference pointing to a 1st level node, that is the root of a sub-tree representing a MOF entity and that could not be moved. In this case, we do not have any issue;
- move/add: again, no violations. However, we are faced with a probable context issue: the addition of a node in a moved sub-tree could probably mean two different desired solutions on the part of the two developers. We should create a warning.

There are no more cases of violations or context issues between two changes in this solution. In the case of the second statement, we mean that even though there could be other context issues (as mentioned before), we can find or handle only the more probable ones deductible during the analysis of changes.

This solution handles the moves, but it is recommended to be used with a certain serialization pattern and, preferably, when we know that the metamodel is the class diagram (we didn't have the chance to test it on others diagrams), due to the various assumptions made before. As we will see in the next paragraph, the third solution has to be checked and verified more carefully: consequently, this could be an acceptable solution if we respect the assumptions.

We also provided a **third solution** which is supposed to deal successfully with nested moves. However, it is a solution that should be further verified, since we had no time to cover all possible situations which could be many and complex. The solution consists of adding some rules to handle nested moves and their interaction with other composite operations. For example, we have to handle the case in which in the version `V` a node `X` has been moved under a node `Y` which, in turn, has been moved under `X` in the other version `V'`. Clearly, no such situation occurred in the previously adopted solutions because in those cases we avoided nested moves. This is a conflict, and, since we cannot resolve it, we should use alternatives or warnings. The problem is that the moved sub-tree may contain nested changes (also other moves), and applying alternatives could lead to an explosion of them. In fact, suppose we have 3 nested alternatives: the higher alternative duplicates all sub-trees representing 2 options. Then the second alternative has to duplicate a sub-tree within the already duplicated sub-tree: consequently, we have 4 options for this alternative (not only 2). Follows that, with the third change, we will have 8 options and so on, following the power of 2. Therefore, we should choose warnings or a different strategy for alternatives, for example avoiding the duplication of them. However, this is a problem if there is an extension tag (as we will see in §3.2.5) and when we want to refer to something which is not XMI-reachable (because it is inside the other alternative tag). We have already analyzed some examples and we have found some similar solutions. Moreover, some further problems will be discussed in chapter §3.2.5 concerning change application.

Summing up, we can consider these solutions a preliminary result: if well tuned, they handle some particular cases (but probably not all of them). Furthermore, they could be used as a hint to review the whole method.

There might be further **context issues**, for example when we discard a deletion. In this case, a deletion is discarded but the deletion or update of the references, that previously pointed to the deleted element, are not discarded. This could create a problem, because we ignore which other changes were related to this deletion: the only thing we can do is to warn the user that there might be related changes, like updates/deletions of connected references.

In this case we warn about a context issue that involves the discard of a deletion that could be related to its close references, deducing their relation from their proximity (in fact they were previously directly connected). So we choose to recognize the context claiming that if they are close, they are probably related, even though we cannot be sure about the existence of such a real relationship. Then we decide to create a warning. However, there could be other related changes, and there could be other strategies to make suppositions about their relationship. We consider only those references which were connected to the deleted node, so we use a distance-1 criterion.

We have not found more methods based only on the XMI syntax to deduce more probable context-relationship between two changes with enough certainty. Besides, we cannot warn about everything that could be remotely connected because that way the result may confuse the user with too many irrelevant suppositions. At this point, we propose a direction for future research on the representation of warnings, which could be somehow included (although it is not very probable) and prioritized with some mechanism. This way, a user can choose to browse only the more probably related changes (for example those based on proximity) or to check deeply those changes that have less chance to be connected. However, as stated at the beginning of this section, finding all these related changes is a widely open issue.

3.2.4 Merge rules

As explained in the previous sections, whenever a conflict, a violation or a context issue arose, we have handled them to avoid a loss of information. We mentioned also solutions, and we will follow some basic rules to be applied in some situations.

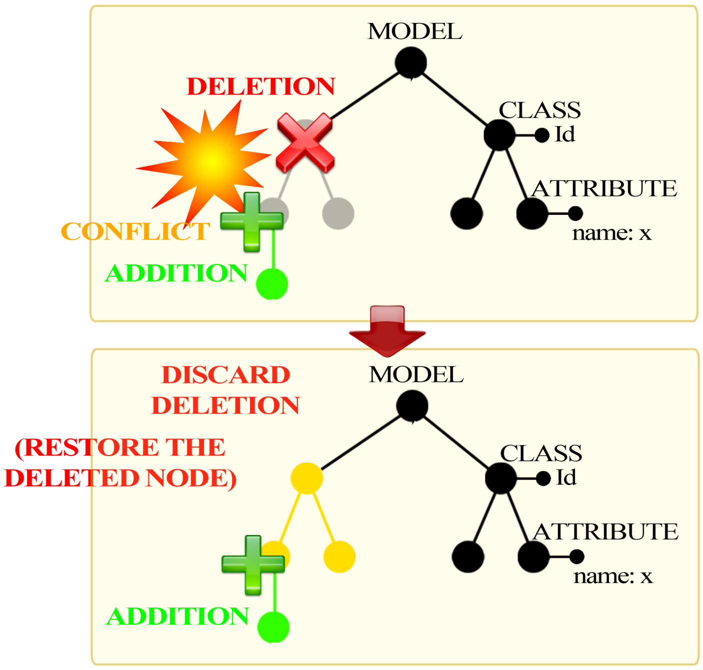


Figure 14: Merge rule: we have to restore the deleted node in order to represent the simultaneous addition

The first cause of data loss is the deletion change. In fact, when such a change is performed and it could affect another change, (e.g. in the case of a conflict, a violation or a probable context issue), we should warn the user about the information that he is losing by the simultaneous application of such changes. The only way to do this is to represent the whole deleted sub-tree somehow in the batch file. In conflicts and violations, we also have to discard the deletion effectively, since in a deletion-conflict we have to represent the other change (update or addition) that has to be applied inside the deleted sub-tree. In the violation, as discussed before, we can opt for correcting the syntax error, but the discard of a deletion does not cause any loss of data (apart from the non-application of the deletion itself, which could be handled by a warning, specifying which sub-tree was supposed to be deleted). In the context issue, we do not need to discard the deletion. However, to inform the user about the hypothetical context-related problem, we should represent what has been deleted, which is the entire sub-tree. To do that, we can represent such an information somehow: however, the simplest solution is, again, to discard the deletion and create a warning (the same thing that we do in other conflicts and violations which involve a deletion), instead of creating a new rule that does the same thing but in a different way. Thus, we have a **unique merge rule** to handle the deletion when it affects other changes: discarding it and creating a warning (Figure 14).

Note that we could also have used a mechanism of alternatives, creating two options which represent the void option of the deletion and the modified sub-tree as the other option. However, this solution seemed to create problems when an option of this alternative overlaps with an option of another alternative (like of a move). The problem consists in representing them clearly, so we decided to follow the discard way, since it does not cause loss of information and it does not show representational issues.

For the update/update conflict, as we mentioned above, a property has been changed: we cannot represent two values of the same property, so we cannot apply them. As we will see in the next section we have to find a way to represent both changes in the same file. We are speaking about an alternative mechanism, that allows us to represent two different options for the same element (to be XMI compliant).

We have another conflict to manage: when we have two moves in which a node `X` is moved under a node `Y` in `Vv` and the node `Y` is moved under the node `X` in `Vv'`. This situation could not be represented: the best thing to do is to discard changes and to insert a warning about their conflict. Another solution could be to adopt the same mechanism used for the conflicts (as we will see later) to represent both possible alternatives.

The rules presented above are exhaustive, i.e. they include all violations and conflicts that need to be handled. Further changes are applied modifying directly the `CA` and applying the change. Since conflicts and violations are already dealt with, every other change could be performed. The only thing we should consider is that it is safer to apply first every change which is not a move, and only then to proceed with moves. This rule is necessary, because all the registered changes are recorded with respect to the `CA`. Since we saw that applying a move before or after other changes does not influence the result (see the conflict detection §3.2.2), we can apply first all the changes modifying the `CA` and then we can apply the moves. We will explain this more in detail in the next section.

Note that the number of rules is rather small because when performing a batch merge, we want to record the widest possible amount of information about conflicts, violations and context issues without taking decisions instead of the user. The purpose of this batch merge is, in fact, to help the user understand relationships between related changes and then to facilitate the manual merge (or using other tools to be implemented), rather then to perform a completely automatized merge (which would require at least a huge component of Artificial Intelligence relying on a large set of information that we do not have) [8].

3.2.5 Creating the merged file

The final step of the merging process is to create the merged file. We have to apply the changes to the `CA` and insert annotations about changes, conflicts, violations and potential problems.

First of all, we proceed with the **application** of all changes that do not provoke a conflict. We create a copy of the `CA`, then we can simply modify it. We can gain access to the changed element by finding the path described on the change statement and then by applying the change. For an addition we create a new substructure identical to the one we have in `Vv` or `Vv'`. In the case of properties, we delete, add or change the value. We have to be very careful about the combination of deletions and moves. In fact, if we apply a deletion and we have previously moved a child node (which is not considered a conflict or a violation, because the result is valid) we lose trace of the source of the node. Even though we have the entire deleted sub-structure in an annotation (as we will see later), it is safer to apply the move before the deletion. But delaying deletions, in case a child-node, placed below the moved node, is deleted, we are not able to reach it by its path. Thus, the question about which one is preferable to be applied first emerges. We choose to apply these changes in a **bottom-up** way: first the changes in the lower nodes are inserted, so that their path will not be modified by a higher level node move or deletion. Notice that by applying the addition and the updates before the moves, we avoid many problems of the same type. In fact, suppose that we have an added node `X` and a moved node `Y` under `X`: if we did not apply the addition of `X`, such a node would not exist, and it would be impossible to apply the move. Furthermore, applying additions and updates before moves avoids the path problem explained in the case of deletion. We do not need to be careful about a move whose source is placed below an added node or a property, since these cases are impossible. Doing these operations is quite easy using an XML parser like DOM, so we decided not to explain further details about it.

During the application of changes, we would also like to mark them with **annotations**: the aim is to show the developer which part of the document has been changed and how. To represent the whole information, we have to report both the modified and the original piece of XMI. We stated that we mark changes "during" and not "after" the application, because, to mark a change, we use a path strategy and we could encounter the same problems as the ones on the application. An annotation should show, as changed, only the latest element in the change statement, which is also the smallest and deepest element changed in the XMI tree hierarchy. For example, if we are speaking about updates, we should highlight only the changed property. If we have a deletion or other composite changes, we should mark the root node of the interested sub-structure. In details:

- addition: we can mark the root node of the added sub-structure or we could put a mark of starting and one of ending. The first solution seems to be the best, since we can put a mark element beside the structure without modifying the original XMI tree. The second solution is more readable, which means that by looking at the XMI document it is more visually clear which part has been changed. Furthermore, the first approach could be tricky since it could happen that a node is added and another sub-tree is moved below it. Thus, marking only the first added node means that we are marking also the moved sub-structure. However, since we mark the moved node as well, in the end, it will be easy to deduce changes anyway. When we add a property, we have no such problems;
- update: as we have previously discussed in the change-detection section (§3.2.1), every update ends with the modification of a property. Thus, we

can create an annotation which points to the property and explains what has been changed (we recall the fact that to reach a property we need the parent-node in the path). We should insert a field into the annotation to show the previous value, in order to avoid loss of information: the user may need to know about it to resolve another conflict;

- deletion: here we have two choices. The first is not to apply the deletion and mark the node as “to be deleted”; the second consists of deleting the node and adding an annotation containing the whole deleted structure, in order to avoid a loss of information about the change. To prevent confusing the user, we prefer to choose the second solution. Since we do not have the node anymore, we should put a reference that includes the parent node as well. For example, if we want to say that the node `Y` has been deleted from the parent node `X`, we should not refer only to `Y` but we need to refer to `X.Y`;
- move: to highlight this change we need an annotation that refers to the moved node, as well as to the changed parent nodes. For example, we should claim in the note that we have moved the node `Y` from `X` to `Z`. This could create a problem when the node `X` is deleted. However, we saw that the deleted node is available in the deletion annotation. Moreover, we create a warning in this case (see conflict detection §3.2.2), so the change can be entirely recognized.

Furthermore, we should associate every change to its author: thus, we need to put this information in the batch file, enabling the merge-user to know which changes are connected by the same “owner”. This is an information that should be represented in all annotations (including alternatives and warnings which will be explained later).

To highlight changes, we need a mechanism. Unfortunately, XMI does not provide it, so we have to use an expedient. We will use this term to define a way that is useful or necessary for our particular purpose, but not always following completely the existing rules. We have two possibilities: using a comment (like in a text-based merge) or using the XMI *extension* element. We will discuss such possibilities at the end of this chapter, since we have to deal with other kinds of additional notes (alternatives and warnings) which need the same representation.

We have spoken many times about creating alternatives and warnings, so we need to define these mechanisms in details: which requirements do they have to satisfy and which are the main related issues. We start with the alternative, then we will explain the warning.

An **alternative** represents a set of options for the same element. Since we are speaking about the comparison of two different versions with a common ancestor (3-way merge), there might be only two possible options represented with respect to the original one. However, we could have more then only two alternatives at our disposal to represent the correct result, since we may combine alternatives creating more options. In the followings, we will deal with two alternatives for the sake of simplicity, but the mechanism could be easily extended to show more options. Creating an alternative means that the same element should be duplicated in order to represent differences. Since we have elements that are recognized by their Id or name, we cannot duplicate them, otherwise we lose the possibility of reaching them uniquely. We could create two new elements which are of the same type as the one we want to duplicate, which would mean that they have a new Id. Note that when we have a name (like in XMI properties), there is a problem concerning the duplication of such an identifier. Furthermore, and more importantly, how could the user know that they are alternatives of an element and they are not simply new independent elements? We should mark them somehow, but we need a mechanism that does not break the language syntax (as for annotations. In our case such a language is XMI). Otherwise, we could use a new and different element (an appropriate alternative element) that should refer to the element that has to be represented by the alternative and its options. In both examples, as we have shown also in the case of annotations, if we want to mark alternatives without breaking syntax at the same time, we need a language support (for example from XMI) like an appropriate metamodel that “understands” alternatives. Otherwise, we need to use some expedients (as we said before, we use this term to define a way that is useful or necessary for our particular purpose, but not always following completely the existing rules). For example, in text-based merge tools such alternatives were performed commenting the same duplicated piece of text (line or lines) representing both options and marking the comment somehow (often with special character sequences). As the next chapter will illustrate, using XMI comments could be a solution to implement an alternative, but we also provide another solution using the XMI tag *extension*. However, it is still a non standard mechanism, since it has not been created to represent alternatives and it does not provide most of the specified fields (described below). Consequently, some requirements need to be satisfied. In fact, analyzing alternatives, we found some requirements to be satisfied when implementing a (generic) alternative mechanism in a structured data file like an XMI document (tree). The alternative element could be composed by more separate parts (for example a nested move) that could be dislocated in different places of the structure. Thus, an alternative element should have:

- an Id: every alternative should be uniquely identifiable. Every sub-structure that belongs to the same alternative should have such an attribute;
- an option Id: for the same reason described above, every fragment belonging to the same option should have this Id to be put together with the others. This way the user (or a hypothetical tools) could clearly see what belongs to the whole option;
- an author Id: we should show the user the authors of each option;
- a difference marker: sometimes we might need to represent the whole element that has been changed in the alternative (for example, if the value of a property is in conflict, we duplicate the whole property but we mark only the value with this tag). Marking the effective part that has been changed could be useful for the user or for a tool to read the differences (e.g. in the case of property we could mark only the value as changed);
- a position mechanism: sometimes, we do not want to duplicate a changed element but its different position in the structure. To represent that, we can duplicate the two options in different places. Otherwise, we could leave the sub-structure choosing one solution (for example the original one in the `CA)` and find a way to say that the root could be placed in two different places (to avoid duplication of a whole sub-structure).

We choose to use alternatives only to represent options for a conflict of the type update/update on the same property. The main aim is to avoid situations in which we may have overlapping alternatives. There is no problem having alternatives for atomic changes (involving properties): they cannot overlap each other, since they are independent and they do not have any part in common. On the contrary, we have some problems using alternatives on composite changes. In fact, if two composite changes need an alternative representation, it could happen that one or more fragment, which should be represented in an alternative, appear in the other one as well. This leads to a very complex representation which could create confusion. Furthermore, such alternatives on composite changes are not very realistic: probably the user will not choose either of them, but he will create a new solution ad hoc [8]. Our main task then is to let him know which are the problems to solve instead of dealing with them ourselves, since we do not have enough information. To do that, we can use a more appropriate mechanism, described below: the warning.

A **warning** is a mechanism whose aim is to show a problem that involves (or may involve, in the case of probable context-related problems) two changes. The difference between the warning and the alternative is that the warning does not propose a solution, but just flags and describes a (hypothetical) problem. We use it widely in the majority of the cases described above because the information that we have, using only XMI, is not enough to deduce a limited set of reasonable options (apart from property conflicts). In the followings, we show some required elements that should be included in the definition of warning:

- an Id: sometimes it could be useful to refer to other conflicts and reach them uniquely;
- an author Id: we should show the user the authors of each option;
- two (or more) change references/descriptions: if we have a set of saved changes on the batch merge or if we have marked them within the original elements (in other words we are sure that all information about changes is reachable by identifier in the merge file) we would use references to connect the involved changes. Otherwise, we need some sort of language to represent appropriately the changes, in order to explain exactly to the

user (or to a tool) which changes are involved. In this thesis we use the change-detection mechanism described in sections §3.1.2 and §3.2.1. Thus, for example, the update with the value `v` of a property `p` of the node `Y` belonging to the sub-structure `X` will be described as the statement `[X, up(Y, up(p, up(v)))]`. Suppose we detect a conflict, a violation or a probable context-related issue with another change: for example, a reference `r`, placed within the sub-tree with root `Z`, which is the child-node of a node `W`, that now points to `X` instead of another sub-structure `S`. We will also have the description `[W, up(Z, up(r, up(S→X)))]`, together with the previous one. With this pair of descriptions placed inside the conflict element, we have the information to highlight all what we want to attract the user's attention (or that of the tool);

- a priority mechanism: this is rather a desired component than a requirement. It might prove useful to distinguish an important problem (for example involving a conflict) from a notice referring to a probable context-problem. The way to implement such a mechanism should reflect how crucial the warning is: for example, in this thesis we may use the priority mechanism with three different values to flag conflicts, violations and context issue;
- an “explanation” field: it is important to explain properly the problems that have been detected, for example, if there was any conflict or violation or if we discarded some of the changes described. It could also explain why we create the conflict, for example when we discard a deletion because it causes a violation with a reference update. In this thesis we do not discuss the way to represent such a field, we simply use the natural language for the explanations.

As mentioned in connection with alternatives and annotations, XMI does not provide a warning mechanism, so we have to use the same expedient. At this point we have to discuss which expedients are available in XMI and which one do we prefer. We identify two possibilities: inserting a comment, like in a text-based merge file, or using the XMI tag extension.

In the case of **comments**, we can simply write notes as we wish, using XML format or even a natural language. The main problem is that such comments are not distinguishable from others. To prevent this problem, we should put some kind of special character sequence to show that we are not dealing with a common comment, but it represents a merge note.

What we have found interesting in the tag **extension**, is that, according to XMI, we can use a special attribute that makes the element ("wrapped" by this tag) an extension of another. This satisfies a requirement described before, in which we desire to create annotations that refer to nodes. For example, if we have to represent an added node, we can add an *extension* element pointing to it. The *extension* tag, since it was created to support interoperability, allows us to specify which tool we are using: this could be useful, since we can simply find a string to define every *extension* element as belonging to a "batch merge tool". This way we have a mechanism to formally distinguish the merge elements we added from other elements inserted by other tools. Finally, every extension element has its own Id, which is a good way to reach them. We have problems when we have to mark a property (which has no Id), but it could be solved simply by marking the parent-node (we need to mark it anyway, since a property is reachable only by its parent-node). Unfortunately, there are no more positive features, so we have no other way to represent further information using standards. This is due to the fact that the XMI language lacks the definition of a mechanism to handle annotations, alternatives and warnings. The main reason is that the batch method for merging models (and generally structured data) is not so widespread, thus there are no standards to represent such mechanisms. The best solution might be a standard definition: once we have a batch merged file, it could be processed and elaborated by other tools created separately and relying on such a standard. This is a way to separate the different tasks of merging, interpreting or visualizing results [8].

Chapter 4

Algorithm

We propose an algorithm which implements the merge process described before. The abstract algorithm is expressed in natural language to simplify its reading. The following instructions are meant to cover all the serialization patterns used by XMI. However, as we discussed before, it works very well if we have no moves at all (described before as the first solution). It works properly if we have a pattern without nested MOF classes and thus a few amount of moves, especially involving the second level of nodes (usually class diagrams). We did not have the chance to test the algorithm thoroughly on the remaining pattern (nested MOF classes and frequent moves of nodes), conseqently, we cannot assure a correct result in a very complex combination of various changes (there may occur problems in the application of changes and in the representation of alternatives and warnings). The algorithm is annotated with comments which explain the reasons for the choices made.

◊ COLLECTING CHANGES (1)
  - find the MODEL node *(we call it R as root)* in the XMI tree;
  - for each child-node E *(we choose E for "MOF Entities")* of R do:
    - **(a)** if E has been added or deleted then report in CHANGES (2)
      - if E has been added and deleted at the same time report in MOVES (3)

- **(b)** if an XML property XP or a REF of E is deleted, added or changed then report in CHANGES (4)
- for each child-node E' of E do the same 2 steps (a) and (b) and so on until the leaves;
- for each M in MOVES do the same steps (a) and (b), taking E as the root of the sub-tree instead of R, and keeping the prefix related to the CA (not to the prefix after the move) (5)

Comments:

1. This whole set of instructions is meant do be executed on both changed versions Vv and Vv' with respect to the CA in a non-deterministic order.
2. The CHANGE set contains all the changes: every change is structured as described in section §3.2.1 dealing with change-detection. For example, if a node X has been deleted we have […path…X, del()]
3. It reports the different parents. It is explained in details in §3.2.1
4. If it has been changed, then it reports how, for example the new node pointed by the reference
5. As said in §3.2.1, whenever we have a move of a node N, this should be matched with the original one placed in the CA and we should continue the change detection: otherwise, the whole sub-tree of N is considered simply deleted (whereas it is not) and it will not be compared with the same one belonging to the CA, hiding changes.

◊ CONFLICT DETECTION, MERGE RULES (6)

- for each deletion DEL check its suffix and
    - if there is a node N that is also (in the other version) in a prefix of other updates, additions, it is a destination of a move or of a new/updated/added reference, then remove the DEL and add a report in WARNINGS (7) explaining why it has been discarded; (8)
    - for each reference that previously pointed to the deleted node and now is updated/deleted, report in WARNINGS (9)
    - if there is a node N which is the source of a move, then report in WARNINGS (9)

- for each update UP of an XML property or a reference in V
  - (c) if the same property/reference is changed (with a different value) in V', then remove the UP and report in ALTERNATIVES (10)(11)
    - if the original reference in the CA pointed to a deleted node N in V or in V', then remove the DEL and report it in WARNINGS (12)
  - if the same property/reference is deleted in V', then remove the DEL and report in WARNINGS (8)
- for each added XML property or reference, if they are added in both V and V' , then do the same thing described in the previous step (13)
- for each move M in MOVES of V
  - if there is another M' of the same element in V' (and it is not moved to the same new father-node) then remove M, the deletion and the addition and report in the WARNINGS (14)
  - if there is a reference REF in V' which has been added or updated in a way that now REF points to a node that belongs to the moved sub-tree or to the prefix of the destination of M, then report in the WARNINGS (8)
  - if in the new prefix of the moved node N there is a node A which is moved in V' under a node B that is placed in the suffix of N, then remove both moves and report in WARNINGS (15)

Comments:

6. Sometimes it is necessary to mix them.
7. WARNINGS is a set which contains records as described in § 3.2.5
8. Every deletion conflict, violation or context issue is managed by discarding the deletion, as explained in §3.2.4
9. It recognizes every distance-1 related reference that could be context-connected with the deletion.
10. Here we can put a further and specific algorithm to find conflicts between the two values.
11. ATERNATIVES contains elements as described in §3.2.5: every element (that represent a conflict) has a sub-set of options, extracted from the changed versions.

12. Keeping the original reference to a node N could break the validity if one of the changed versions have deleted N. Then we have to act as when we want to avoid syntax violations. In this case, discarding the deletion also causes the warnings about connected references. Since all these operations are caused by the initial conflict (c), we should report the cause in the case of every element inserted in WARNING.
13. This situation corresponds to the situations in which the same property has been changed.
14. Conflict due to the move of the same node. As mentioned in §3.2.2 and §3.2.3, this conflict could not be resolved and we can discard both moves inserting a warning, or represent them as alternatives: the latter representation is more visual, but it could lead to inconsistencies with other nested move conflicts.
15. Useful only for those patterns which has nested moves.

◊ CHANGE APPLICATION, ANNOTATIONS, WARNINGS AND ALTERNATIVES

- copy the whole CA in a new file MERGE (16)
- for each addition in CHANGES, it is performed in MERGE
    - mark the new nodes as added
- for each update in CHANGES, the property is changed in MERGE
- for each alternative in ALTERNATIVES
    - create two duplicates of the original element and apply the changes separately
        - if there is no original duplicated element (two additions) then choose non deterministically one of the two options and apply it (17)
    - “wrap” the two options using the comment or the extension mechanism
    - refer to the original element (or the applied one in the case of two additions)
- for each move in MOVES and deletion in CHANGES apply them using a bottom-up strategy (18)
- for each warning in WARNINGS create the extension sub-tree (or a comment) referring to the involved nodes.

Comments:

16. Since we saved the changes with respect to the CA, we need to duplicate and modify it with them.
17. In the case of an add/add of the same property we have nothing to refer to (there is not an original property in the CA). Then we apply one of them and we use the other as alternative. This is the only case in which we do not respect the symmetry constraint, but we should consider that it is a very rare situation. Furthermore, it does not cause any problem.
18. As mentioned in §3.2.5.

Chapter 5 Implementation

In this chapter we propose a practical application of the theoretical principles discussed in the previous chapters.

We opted for creating a tool which elaborates three models serialized by the UML editor ArgoUML. The reason for this preference is mainly that ArgoUML provides the fundamental requirement (see §3.1.1) of keeping the Ids unchanged throughout the saving process. Besides, ArgoUML uses the first serialization pattern described in §, which means that we did not have to handle the move changes (although the tool is extendible so that this feature could be added later on).

The tool performs a **virtual merge** [8] which consists of the first three steps of the whole merge process with the addition of a graphical interface that shows the changes and the detected problems. Among these, the tool displays all conflicts, reference violations and two kind of context-related problems. The tool is supposed to provide an initial framework which could be easily expanded.

We have created various classes which are contained in the `it.unipr.XMIMerge` package.

5.1 Loading files

First, those files must be loaded whose virtual merge has to be shown. You can load directly XMI files exporting them from the editor, or you can directly load the ArgoUML project files with the extension `.zargo`. In fact, it is simply a container which stores files (compressed with zip) describing the model: among these, we can also find the XMI file. Thus, we can extract the XMI automatically, (without requiring the user additional step of exporting from the editor) loading the project that he saved during the development of the model. This is a function implemented in the class `FileManager` in the `it.unipr.XMIMerge.io` package. It includes two methods:

- `showFileChooser`, which allows the user to open the file he prefer (he can choose `.xmi` or `.zargo`) using the Swing libraries, and
- `unzipFileIfNecessary` which extracts the XMI part if we have chosen to open a project developed in ArgoUML.

The first method is repeated three times at the beginning of the execution of the program. In fact, the user has to choose three files: the `Common Ancestor` and the two modified versions. It is important that the first file is the `CA`, while the other two files can be selecting according to the user's order of preference (an order which is then used to show differences between the models). The second observation implies that the appearing result will not depend on the order in which we open the different versions, which is an important feature to ensure that the result is the same every time the program is launched with the same input files in a different order (satisfying the symmetry requirement stated in §3.1.5).

5.2 Parsing models

Once we uploaded the files they have to be interpreted as XML, and we used `JDOM` libraries to perform the parsing. For every file a `SAXBuilder` is created, which produces `Documents` enabling the browsing of the XML tree. Thus, it was necessary to create the structures that represent the abstract trees of MOF type to be compared subsequently. Not every node is important for the logical description of the model: for example, we are interested only in the contents of the XML subtree, named by the tag model. The structure represents a tree (`MOFModel`) which contains a root node, a `hashmap` to reach its nodes with their `Id` and a set of all references contained in XMI. The nodes are instances of `MOFNode`, a class dedicated to the representation of each element of the model, in which the Ids are saved: the father, children, properties and references. This allows us to browse the MOF tree easily. In addition, the `MOFModel` provides some functions which calculate the path of the nodes (`getPrefix`) and their descendants (`getDescendants`). As we will see, these two functions may be useful indeed.

The parsing is operated by the `XMI2MOFTranslator` class through the `parseXMItoModel` method. First, the *`model`* element is identified, then the subtree (which has *`model`* as root) is visited in pre-order: every time we find a node with an Id, a corresponding `MOFNode` is created. The XMI format of ArgoUML includes nodes without Id, used to "encapsulate" the other nodes (containing semantic information possibly useful for the internal editor). We emphasize, in fact, that XMI is defined so as to be extensible and customizable if necessary, thus, we have to manage these nodes. The `XMI2MOFTranslator` class provides two methods, one is "verbose", the other not:

- The first creates `MOF nodes` for the nodes without `Id`, as if they were elements of the model: a new `Id` is created concatenating the `Id` of the father (which is unique) and the name of the XML tag. Since the `Id` must be unique, we must make sure that also the one created by the tool is that way: it was proved that in ArgoUML sibling nodes never have the same name (whilst they may have the same name to those of other elements created as children of different nodes). Thus, concatenating the (unique) `Id` of the parent with the name of the child (unique among the siblings) we obtain again a unique `Id`;
- The second ignores the nodes without `Id`, building a simpler structure MOF. Sometimes it can cause problems if the nodes without `Id` contain

references (they would be lost, it must be fixed).

Once you've created three MOF models, they are ready to be compared to find those differences that concern us.

5.3 Matching models

The matching of models, thanks to the Ids, is pretty simple. In fact, ArgoUML keeps the `Id` of the items that have been changed throughout the saving process. This means that if you open a model with an `E` element with the `Id I`, this will be saved at the end of the session with the same `I`. This is obviously essential for the identification of nodes.

The only case when this does not happen is when we change the items of the type `Multeplicity`. These are items that are always close to the leaves (or they are leaves themselves) and they are re-created (resulting in a change of `Id`) every time one of their properties is changed in the editor. This does not present a serious problem, since we are able to recognize and identify them as "changed" instead of "deleted and added" ones (as we can limit the occurrence of the problem only for this type of elements).

Having the `Id`, we can then refer to the same element in three different models to investigate the changes in which it is involved, and, thus, provide the differences between the modified versions and the `CA`.

5.4 Identification of changes

We can start from the root of the tree described by the `CA` (and necessarily shared by the other two models as they are derived from the modification of the `CA`) and visit it in pre-order, gathering in an instance of the class `Difference` the set of changes. We compare every element we find with the one of the modified version which has the same name. The following three situations may arise then:

- an element in the `CA` is not detected in `Vx` (the procedure is identical for both `V1` and `V2`, so we use `Vx` to identify the version that is being analyzed);
- in `Vx` an item which was not present in the `CA` is detected;
- a property (including references) of the same node has been changed.

In all cases an instance of the class `Change` is created:

- a unique Id is created (so that we can refer to it in an unambiguous way later on),
- the type of the operation is set which is identified by static variables belonging to the class:
  - `ADD` - if an element has been added to `Vx`;
  - `DELETE` - if an element has been deleted from `Vx`;
  - `UPDATE` - if the value of a node property has been altered.

We can observe that the operations have a "direction", ie we have to interpret them as "all what happened to the `CA` to obtain `Vx`". For example, if there has been an `ADD`, it is understood as an addition of the node `N` to the parent node `F` in `Vx`, which means that `N` was not present as a child of `F` in `CA`. This will be important later on to determine whether two *changes* from two different `Difference` might create problems;

- the type of element involved is set (node, property or reference);
- it creates a list of nodes representing the path from the root to the node involved in the *change*. This can prove to be very important, as we shall see, when we will search for problems;
- in the case of a property or a reference, the new and the old value is memorized: this may also be useful in further investigations of context issues.

This is carried out by the class `Differentiator` with the method `diff`, which returns an instance of `Difference`. This class provides access to various subsets of changes, for example, with `getUpdates` only those changes are shown which have the value UPDATE as operation. Moreover, it also keeps track of which models are involved in the comparison and which one of them is the CA.

5.5 Problem detection

Once the program discovered the differences, it has two sets of Difference to compare. We can observe that this procedure does not necessarily force us to use only two Difference, as it may as well, insert a problem detector which may involve, for example, more than two versions V1 and V2, enabling us to deduct information or probable context problems in a more accurate way. However, in this first version of the tool we will provide a comparison of two versions only.

At this point, being faced with three types of problems (as described in chapter 3), conflicts, violations and context problems, we decided to create three types of detector (detector) different and disconnected from one another. We need this to keep responsibilities properly separate. Furthermore, it is unlikely that we decide to change the detection of conflicts, whilst it may be more common that we need to add a new context problem for analysis. Thus, we have a ConflictDetector, a XMIViolationDetector and ContextIssuesDetector, which produce, after having scanned the two sets of Difference, a collection of instances of Conflict, XMIViolation and ContextIssue respectively (all classes derived from the generic class MergeProblem). Let's see how they work in detail.

5.5.1 Conflicts

The ConflictDetector generates, given as inputs models and sets of Difference in the right order, a set of hash consisting of Conflict instances. This class contains the information about the conflict, such as the unique Id (between the conflicts) and the Ids referring to the changes which are in conflict. How does the detector find the conflicts? This class has a detect method, which executes in turn three distinct methods:

- `delDetection`: examines all those changes whose operation is a `DELETE` contained in the first `Difference` object (given as argument), and checks if they are in conflict with the changes which are in the second instance of `Difference`. This operation has a direction, i.e. it is necessary to execute the same method twice on the `Differences` passed via input with a reverse order. In fact, we have to check whether also the *deletions* of the second `Difference` cause problems with the *changes* contained in the first one. In particular, the method analyzes the *change* and identifies the deleted node (in case the deletion involved a node and not a property), then it collects all the descendants (deleted as well). Subsequently, it visits all the *changes* of the other `Difference`, analyzing the path of each of them from the root to see if it contains at least one deleted node. In case it finds one, the removal of that node will surely lead to conflict, since a deleted node (or a descendant, or the node itself that we are analyzing) was simultaneously changed (a child node has been added or a property has been changed). In this case an instance of Conflict is created. In case the removal involved only one property, the method checks only if it had also been changed into another *change*.
- `AddAddDetection`: This method deals with finding those conflicts that arise from the addition of the same property but with different values to both versions. In fact, if we add property `P` to node `N` (where the `Id` of `P` is accessible only through the name of the node `N`) with value `T` in version `V1`, but the same property `P` in `V2` with a value `T′` different from `T` is added, a conflict would arise. This because within the same node two properties with the same name cannot coexist, otherwise one of them would not be reachable.
- `UpUpDetection`: it detects conflicts which involve the same property `P` changed simultaneously by the value `T` in `V1` and by `T'` in `V2`. Also in this case an automatic merge would not be possible without losing information about at least one of the two changes.

5.5.2 Violations

The `XMIViolationDetector` works similarly to the previous detector. The biggest problem that can affect the XMI syntax (and which has not been detected as a conflict yet) occurs when a reference points to an object which was deleted. This is not considered a real conflict because the reference and the absence of the referred object can coexist within the file, however, this invalidates the XMI syntax. The main method is once again `detect` and it returns a set of instances of `XMIViolation`. This class, apart from containing an `Id` and references to the changes involved (as `Conflict`), it also contains a field of the `String brokenElement` type, which provides information about the element causing the violation. The primary method used for the detection of violations is:

- `findDeletedReferredElement`: runs all the *changes* of the type *deletions*. It identifies the deleted node and it collects in a set all its descendant nodes (they are deleted as well, of course), then it checks whether in the other version there have been additions or changes concerning the references which, once modified, point to one of the elements contained in the set (they will become Ids of the node). In that case it creates an `XMIViolation`.

We can observe that this method detects a certain type of XMI violation: it may happen that in later versions of XMI, other combinations of elements may invalidate this format without causing a conflict. Then one could simply add a method called by `detect` which identifies this type of problem.

5.5.3 Context problems

Also the class `ContextIssuesDetector` provides a method `detect` to find the context problems. Context problems, as mentioned above, may be infinite, some of them more frequent, more probable or more critical whilst others less so. Above all, though, they require the knowledge of a higher semantic level which is often not detectable in the XMI format. Thus, we can speculate about hypothetical problems, but without having the certainty that they really are problems. In fact, the detection of all validity problems concerning the language used by the model (eg UML) is a field of study which is still open to research. It is not the ultimate goal of this work to explore this field, however, we have created a framework that could allow the evolution of a recognition system for context problems. The class `ContextIssuesDetector`, being a prototype, includes two methods which are able to detect two types of `ContextIssue`, to show how it happens. In the future, it will be possible to simply add methods that the `detect` method can recall in order to find new hypothetical context problems. It is also important that for each detected problem a priority index is associated, which provides an estimate of how problematic the combination of the two changes could actually be. This can prove useful for a developer who wants to select a priority threshold and filter the potential problems shown. An explanation of the two methods created in the tool to detect two different `ContextIssues` may help to understand this clearly. One of the examples is rather generic and has a lower priority, whilst the other one is more specific and has a very high priority. The two methods for these detection are:

- `sameNodeInvolved`

  In this case, the idea is that by changing simultaneously two properties within the same element, the two changes can cause semantic problems. This does not necessarily create a problem: if, for example, we change the property `name` and the property `isAbstract`, it is unlikely that the two changes together create problems. In general, the probability that two properties are connected does not seem very high. Therefore, we chose to assign a (fictitious) priority of `30` out of `100` in order to quantify this probability. This way the user will be aware that the two changes may create a problem, even if it is not very likely that such a situation occurs. Perhaps it may be useful to know about the eventuality in case there is only one potential problem, but what would happen if we had a hundred of them? The user should have the power to avoid analyzing all possible problems, especially the less probable ones.
- `UpperLowerDetection`

> As we can guess from the name, this method tries to find all the elements which contain the properties `upper` and `lower`. Usually these elements are referred to as cardinality (or `multiplicity`), i.e. elements that specify the minimum and the maximum of the occurrence of their father elements (for example, an attribute that can have a minimum of `1` to a maximum of `5` occurrences).

It may happen that in `Vv` the minimum `(lower)` is modified and in `Vv'` the maximum `(upper)` is changed. This would not create a problem in itself, or it could fall in the cases described by the previous method. However, after the two independent changes the maximum may become lower than the minimum, which arises almost always a semantic problem. Being able to detect this particular situation (it is sufficient to check whether one value is greater than the other one), we can detect a combination of changes that creates a context problem with high probability. We can then give such `ContextIssue` a priority of `90` out of `100.` If the user chooses to reduce the number of potential problems and, thus, to raise the threshold below which potential problems do not appear, this kind of problem will often be above such threshold, having more chance of being detected by the user (it would be reasonable given its critical nature).

5.6 Visualization of problems

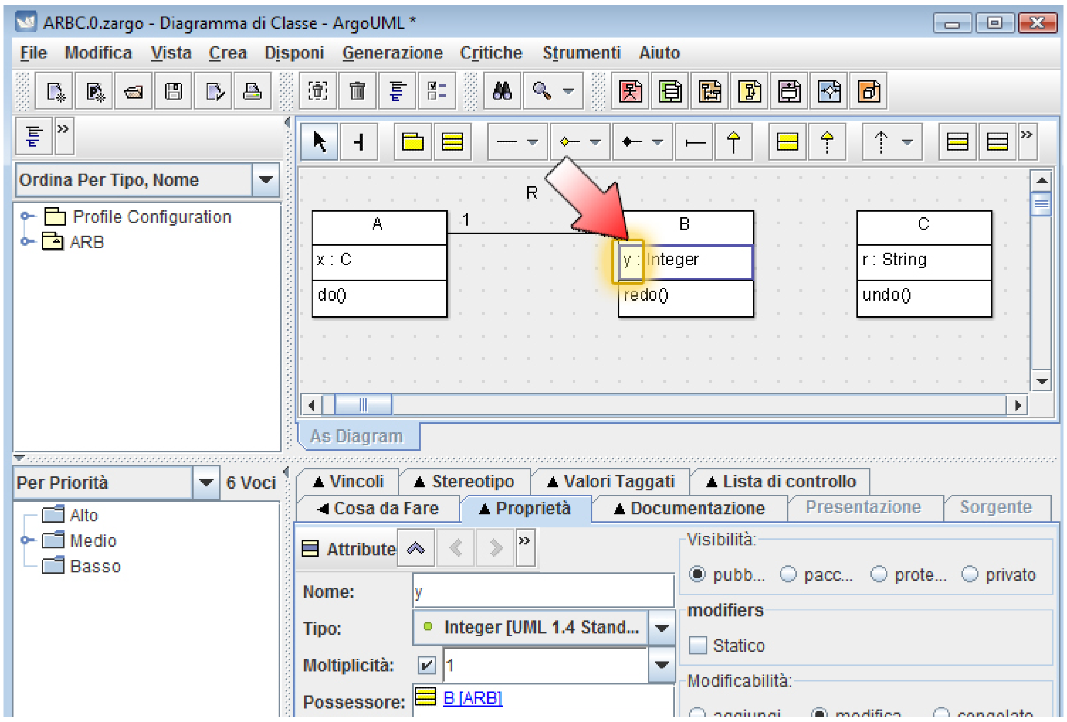
For the visualization of problems, we created an interface that represents the models as trees (the user could see the objects of its model as nodes). Since the XMI format is not an easily readable, we decided to show each element as a node identified by its name and type property. The properties and references are, thus, displayed in subtrees with the same name which belong to each node. Three trees of the three compared models are displayed (on the left the `CA,` in the middle and on the right the two modified versions). First, the nodes changed compared to `CA` are shown in the two new versions (using different symbols to indicate whether they have been added, changed or deleted). This can prove very convenient for the user, who can check all changes to determine whether the problems encountered are related to a set of changes, which, considered together, have a more complex meaning. For example, a refactor consists of more changes at a time, and if one of them creates a conflict, the user is aware of the fact that he has to take into consideration also the others (but, obviously, the program is not). In addition, on the top of the screen, three lists were inserted, one for each type of problem: conflicts, violations and context problems. Here all the identified problems are shown, and it is possible to select them. Then in the two trees, representing the modified versions, those nodes (and thus those elements) will appear highlighted which are involved in the two changes that create a (potential) inconsistency. Under the representation of nodes also a `TextArea` is inserted where a detailed message corresponding to a given problem is displayed (it is inserted in the instance of `MergeProblem` at the time of its creation). We decided to omit the description of the part concerning graphical programming which involves the Java Swing libraries, as it is not strictly relevant to this work. It allows a better visualization, but it does not add any information.

5.7 Example

Follows a very simple example of how the tool works (an update/update conflict). In the appendixes there are more examples which show some other tool features.

We created three models using ArgoUML: the first is the `CA` and the other two were created by modifying the `CA`.

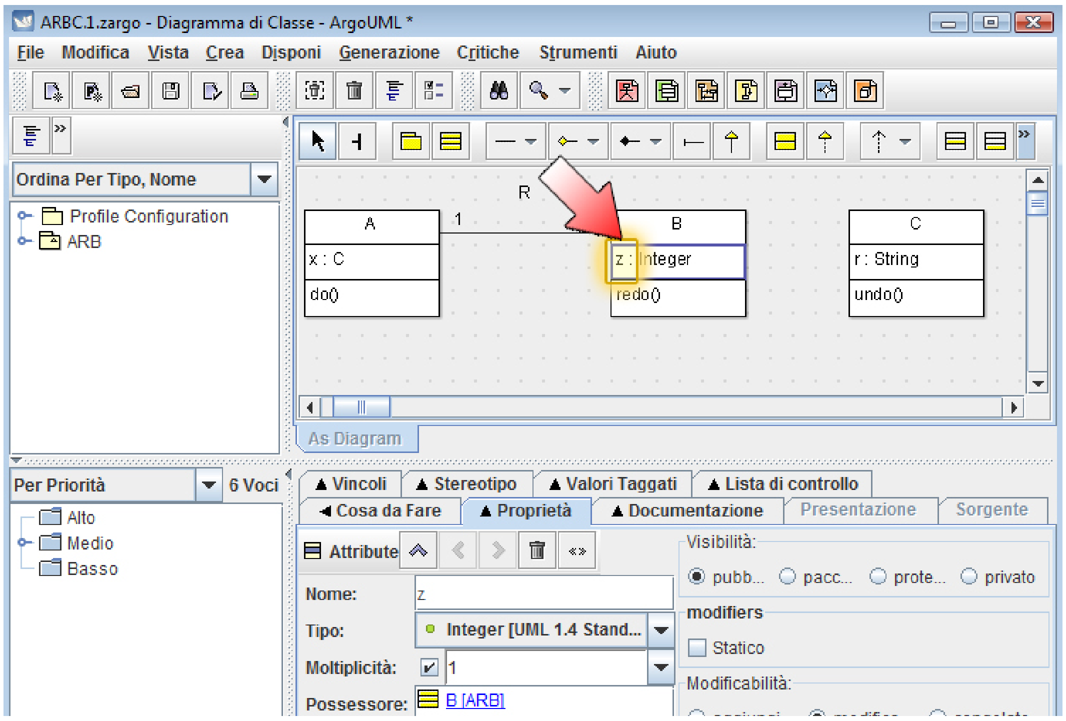
CA:



Nome: y
Tipo: Integer [UML 1.4 Stand...
Moltiplicità: ✓ 1
Possessore: B [ARB]
Visibilità: [✓ pubb...] pacc... prote... privato
Modificabilità: aggiungi [✓ modifica] congelato

Figure 15: *Common Ancestor from ArgoUML*

In version `V1` we find that the attribute's name his name now is `z` instead of `y`:



Nome: z
Tipo: Integer [UML 1.4 Stand...
Moltiplicità: ✓ 1
Possessore: B [ARB]
Visibilità: [✓ pubb...] pacc... prote... privato
Modificabilità: aggiungi [✓ modifica] congelato

Figure 16: *Version 1 in ArgoUML: the name of the attribute y is now z*

Finally, in `V2` the same attribute has been modified as `k`:

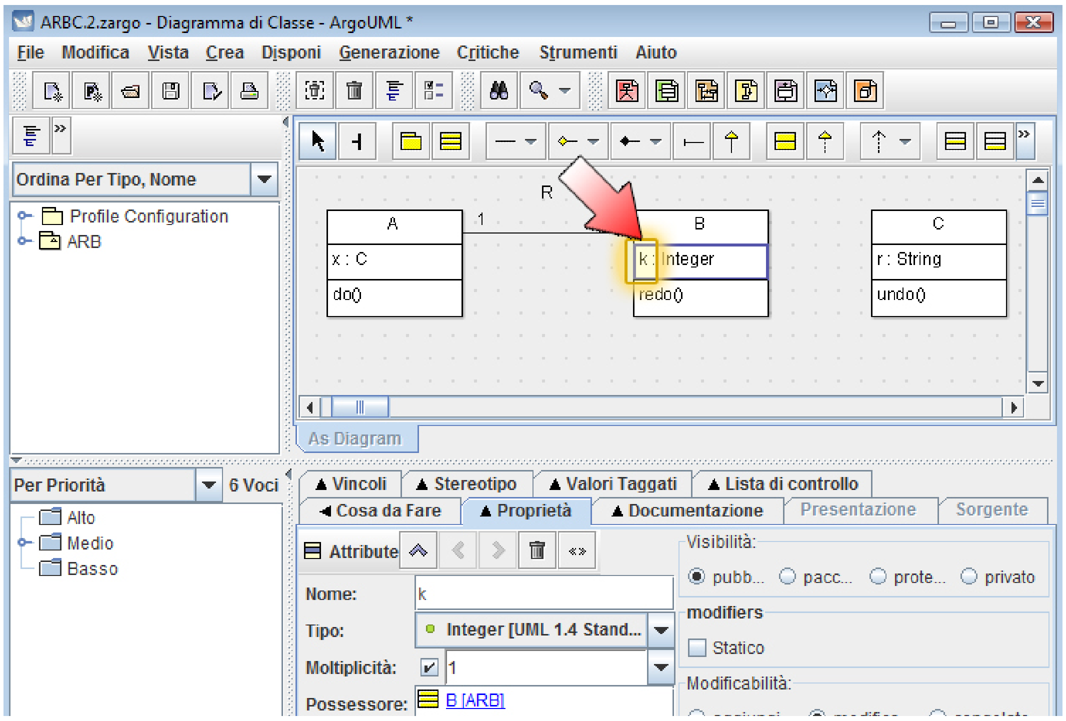
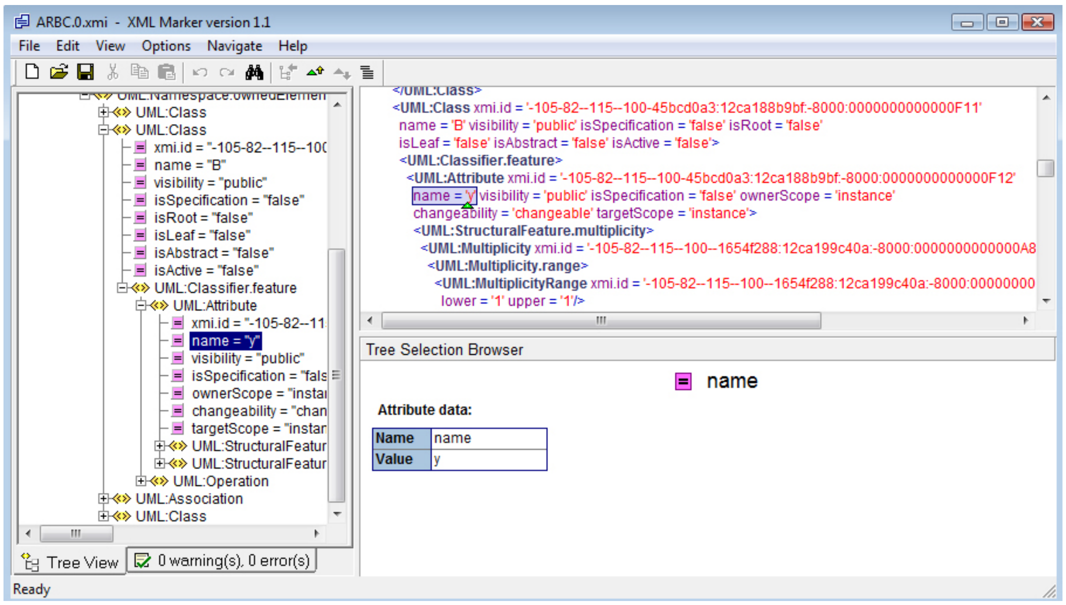


Figure 17: Version 2 in ArgoUML: the name of the attribute y has been changed into k

In the following image we can see the XML attribute named `y` belonging to the `CA`:



```
</UML:Class>
<UML:Class xmi.id = '-105-82--115--100-45bcd0a3:12ca188b9bf:-8000:000000000000F11'
 name = 'B' visibility = 'public' isSpecification = 'false' isRoot = 'false'
 isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
 <UML:Classifier.feature>
  <UML:Attribute xmi.id = '-105-82--115--100-45bcd0a3:12ca188b9bf:-8000:000000000000F12'
   name = 'y' visibility = 'public' isSpecification = 'false' ownerScope = 'instance'
   changeability = 'changeable' targetScope = 'instance'>
   <UML:StructuralFeature.multiplicity>
    <UML:Multiplicity xmi.id = '-105-82--115--100--1654f288:12ca199c40a:-8000:00000000000000A8
     <UML:Multiplicity.range>
      <UML:MultiplicityRange xmi.id = '-105-82--115--100--1654f288:12ca199c40a:-8000:00000000
       lower = '1' upper = '1'/>
```

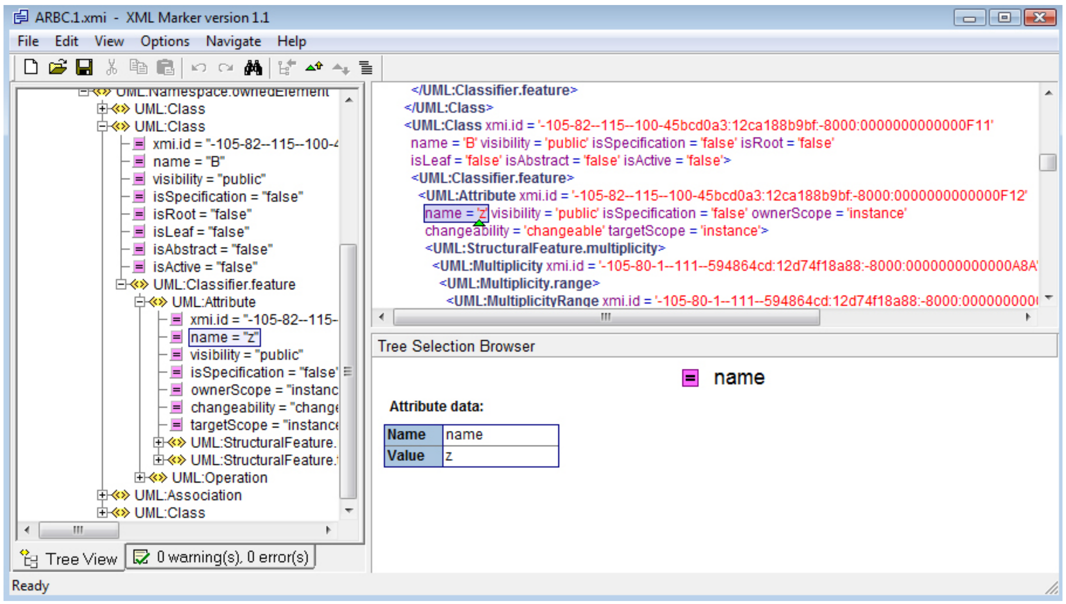
name

Attribute data:

Name	name
Value	y

Figure 18: XMI file of the Common Ancestor

We can observe the same changed attribute in `V1`:



ARBC.1.xmi - XML Marker version 1.1

```
UML:Namespace.ownedElement
  UML:Class
  UML:Class
    xmi.id = "-105-82--115--100-4
    name = "B"
    visibility = "public"
    isSpecification = "false"
    isRoot = "false"
    isLeaf = "false"
    isAbstract = "false"
    isActive = "false"
    UML:Classifier.feature
      UML:Attribute
        xmi.id = "-105-82--115-
        name = "z"
        visibility = "public"
        isSpecification = "false"
        ownerScope = "instanc
        changeability = "change
        targetScope = "instance
        UML:StructuralFeature.
        UML:StructuralFeature.
      UML:Operation
  UML:Association
  UML:Class
```

```
  </UML:Classifier.feature>
 </UML:Class>
 <UML:Class xmi.id = '-105-82--115--100-45bcd0a3:12ca188b9bf:-8000:000000000000F11'
  name = 'B' visibility = 'public' isSpecification = 'false' isRoot = 'false'
  isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
  <UML:Classifier.feature>
   <UML:Attribute xmi.id = '-105-82--115--100-45bcd0a3:12ca188b9bf:-8000:000000000000F12'
    name = 'z' visibility = 'public' isSpecification = 'false' ownerScope = 'instance'
    changeability = 'changeable' targetScope = 'instance'>
    <UML:StructuralFeature.multiplicity>
     <UML:Multiplicity xmi.id = '-105-80-1--111--594864cd:12d74f18a88:-8000:000000000000A8A'
      <UML:Multiplicity.range>
       <UML:MultiplicityRange xmi.id = '-105-80-1--111--594864cd:12d74f18a88:-8000:000000000
```

Tree Selection Browser

name

Attribute data:

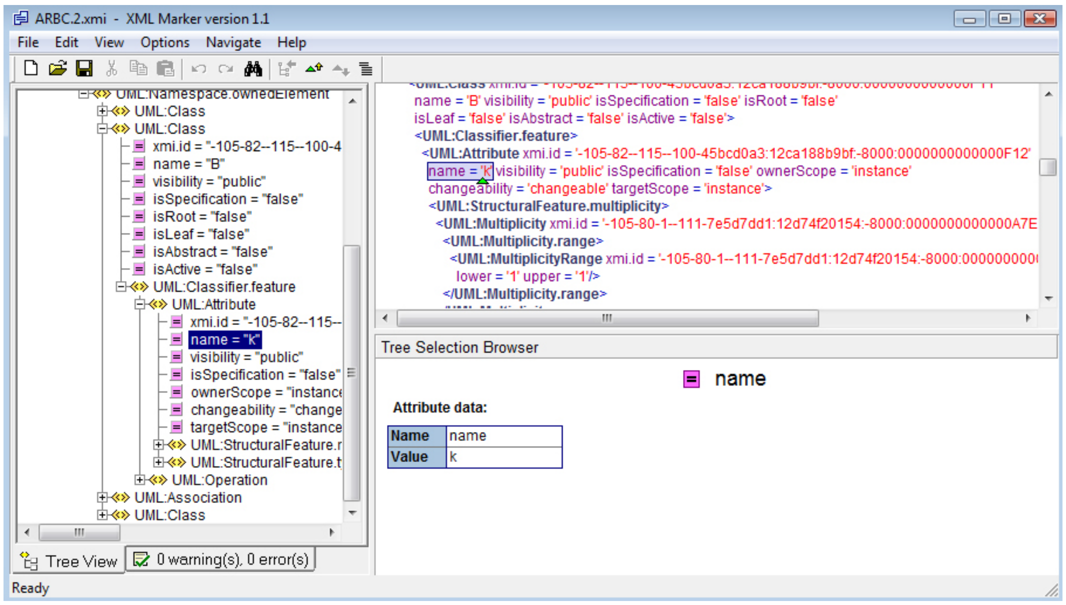
Name	name
Value	z

Tree View | 0 warning(s), 0 error(s)

Ready

Figure 19: XMI file of Version 1

and in `V2`:



ARBC.2.xmi - XML Marker version 1.1

```
UML:Namespace.ownedElement
  UML:Class
  UML:Class
    xmi.id = "-105-82--115--100-4
    name = "B"
    visibility = "public"
    isSpecification = "false"
    isRoot = "false"
    isLeaf = "false"
    isAbstract = "false"
    isActive = "false"
    UML:Classifier.feature
      UML:Attribute
        xmi.id = "-105-82--115--
        name = "k"
        visibility = "public"
        isSpecification = "false"
        ownerScope = "instance
        changeability = "change
        targetScope = "instance
        UML:StructuralFeature.r
        UML:StructuralFeature.t
      UML:Operation
  UML:Association
  UML:Class
```

```
  name = 'B' visibility = 'public' isSpecification = 'false' isRoot = 'false'
  isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
  <UML:Classifier.feature>
   <UML:Attribute xmi.id = '-105-82--115--100-45bcd0a3:12ca188b9bf:-8000:000000000000F12'
    name = 'k' visibility = 'public' isSpecification = 'false' ownerScope = 'instance'
    changeability = 'changeable' targetScope = 'instance'>
    <UML:StructuralFeature.multiplicity>
     <UML:Multiplicity xmi.id = '-105-80-1--111-7e5d7dd1:12d74f20154:-8000:000000000000A7E
      <UML:Multiplicity.range>
       <UML:MultiplicityRange xmi.id = '-105-80-1--111-7e5d7dd1:12d74f20154:-8000:000000000
        lower = '1' upper = '1'/>
      </UML:Multiplicity.range>
```

Tree Selection Browser

name

Attribute data:

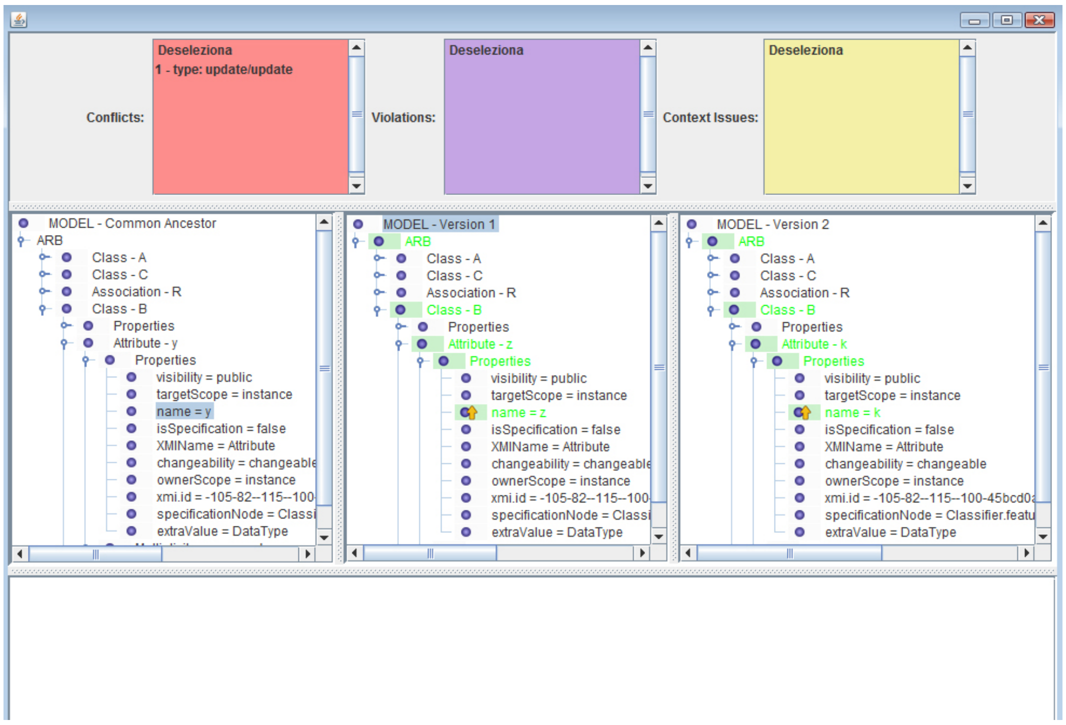
Name	name
Value	k

Tree View | 0 warning(s), 0 error(s)

Ready

Figure 20: XMI file of Version 2

The tool, once we uploaded the files, will show the trees of the three models in the following way: on the left the original (`CA`) appears, and on the right we can see the two versions. The nodes (the elements) that have been modified, respectively, in `V1` and `V2`, are shown in green. At the top we can find the list of problems detected: among the conflicts there is an update/update one:



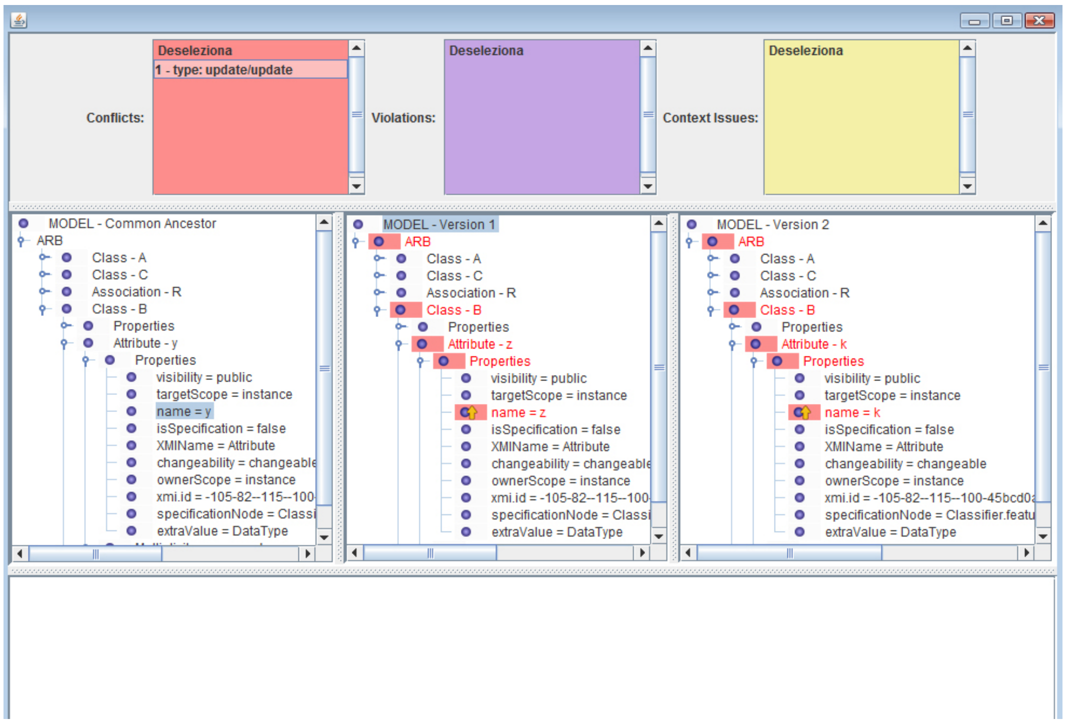
```
MODEL - Common Ancestor
ARB
  Class - A
  Class - C
  Association - R
  Class - B
    Properties
    Attribute - y
      Properties
        visibility = public
        targetScope = instance
        name = y
        isSpecification = false
        XMIName = Attribute
        changeability = changeable
        ownerScope = instance
        xmi.id = -105-82--115--100
        specificationNode = Classi
        extraValue = DataType
```

```
MODEL - Version 1
ARB
  Class - A
  Class - C
  Association - R
  Class - B
    Properties
    Attribute - z
      Properties
        visibility = public
        targetScope = instance
        name = z
        isSpecification = false
        XMIName = Attribute
        changeability = changeable
        ownerScope = instance
        xmi.id = -105-82--115--100
        specificationNode = Classi
        extraValue = DataType
```

```
MODEL - Version 2
ARB
  Class - A
  Class - C
  Association - R
  Class - B
    Properties
    Attribute - k
      Properties
        visibility = public
        targetScope = instance
        name = k
        isSpecification = false
        XMIName = Attribute
        changeability = changeable
        ownerScope = instance
        xmi.id = -105-82--115--100-45bcd0
        specificationNode = Classifier.featu
        extraValue = DataType
```

Figure 21: The tool user interface shows the CA (left) and the changes (green) in V1 (center) and V2 (right)

If we select the conflict, the elements involved appear:



```
MODEL - Common Ancestor
ARB
  Class - A
  Class - C
  Association - R
  Class - B
    Properties
    Attribute - y
      Properties
        visibility = public
        targetScope = instance
        name = y
        isSpecification = false
        XMIName = Attribute
        changeability = changeable
        ownerScope = instance
        xmi.id = -105-82--115--100
        specificationNode = Classi
        extraValue = DataType
```

```
MODEL - Version 1
ARB
  Class - A
  Class - C
  Association - R
  Class - B
    Properties
    Attribute - z
      Properties
        visibility = public
        targetScope = instance
        name = z
        isSpecification = false
        XMIName = Attribute
        changeability = changeable
        ownerScope = instance
        xmi.id = -105-82--115--100
        specificationNode = Classi
        extraValue = DataType
```

```
MODEL - Version 2
ARB
  Class - A
  Class - C
  Association - R
  Class - B
    Properties
    Attribute - k
      Properties
        visibility = public
        targetScope = instance
        name = k
        isSpecification = false
        XMIName = Attribute
        changeability = changeable
        ownerScope = instance
        xmi.id = -105-82--115--100-45bcd0
        specificationNode = Classifier.featu
        extraValue = DataType
```

Figure 22: If we select the conflict, the tool shows (red) the sources of the conflict in both versions

Chapter 6 Discussion

In this chapter we will discuss our work. We have shown that the XMI approach is not supported enough by the XMI standard itself and by tool vendors to perform a model merge. Nevertheless, we have said that it is possible to define a process to handle the task of merging with three XMI files. We will summarize our results and we will specify under which restrictions they hold. Then we will compare our work with other three related ones: an operation based, a formal approach in the model domain and an XML merge algorithm. Finally, we will see how the work could be extended by further research.

6.1 Results and restrictions

In chapter 2 we have seen how XMI shows a non homogeneity in representing models, caused both by the language definition and by the implementations of different tool vendors. Then we have to state that we cannot provide a general merge tool that covers every possible set of XMI files. This is the first (negative) result that emerges from this work. However, by adding some restrictions, like choosing an XMI version and ignoring tool implementations, we have showed that a merge process could be defined to handle the merge task among XMI files. We will present the restrictions and the results we obtain in each part of the process.

We are able to identify every possible change between two versions of a model with the help of the common ancestor: this is possible because for each

XMI element we have a correspondence (provided by the Id) in both changed files. Consequently, we can state that if something has been changed inside the same element, we are able to find it. This is true only if we assume that all the XMI files were serialized with the same pattern of serialization (specification restriction) and if the same id is kept for the same XMI element (implementation restriction) or if a match was provided in advance (environment restriction). However, if the first and one of the other constraints are satisfied, we are able to report all the information about both changed versions in the merged result.

The same restrictions have to hold again to guarantee a conflict detection among changes, since this process depends on the change detection (and generally they have to hold for the whole merge process for the same reason, so we will not repeat this in the next paragraphs). However, such a conflict detection is correct, complete and cheap: we prove the first two statements only informally, since it can be easily deduced from the detection definition in section 3.2.2. In fact, we are faced with a conflict only when the same property is modified, when a change is placed in a deleted sub-tree, or if the same node has been moved. The method of using a dynamic unit of comparison, which follows the branch of the tree in depth, makes sure that we cover every change in every branch, even if there are moves (thanks to identifiers). We come across conflicts in all of the previous cases, so the method is correct and complete. We also have the positive side effect that it is cheap, since when it finds a deletion (and it surely finds the root node of the deleted sub-tree first) it finds all conflicts involving the deleted sub-tree without analyzing it. Moreover, this particular working policy allows us to integrate other algorithms (even if we have not tried it) in order to refine the conflict detection within the leaf value, depending on the different format (for example if we have a piece of code in a node value, we can continue to analyze it, selecting a dedicated text algorithm when we reach it).

In the interpretation part we required also the analyzed files to be XMI valid. This is not a strong restriction, since there is no reason for any editor to serialize an invalid XMI. The good result was, in the case of (XMI syntax) violation detection, to find only those situations in which the separate application of two changes produced two valid files, while their application in the same document violated the syntax. This means that we avoided to process the whole file finding violations, since the part of the document that was not changed remained valid: instead, we found only a small set of such “dangerous” changes (involving references and moves), which had to be checked in order to detect violations. Even though we did not provide a better result than the one performed by an XMI validator, we proposed a cheaper and faster way to discover violations (we do not have to validate the whole document against all XMI rules, but only those dangerous changes). The part dealing with the context-related problems provided, as expected, only a small set of those recognized probable problems that we could encounter at the model level. This is reasonable, since (on such a level) we have only a small amount of information provided by the MOF structure and represented by the XMI tree, which represents a very high abstraction of the model. We provided a mechanism that uses the proximity of changed MOF entities to determine whether they could be related at a higher level. We have not found any other way to deduce related problems without proposing excessively case-related ones. In fact, the problem of detecting related changes is a very complex and open issue even if we know the specific semantic of the model [8], so using XMI we can simply speculate on it.

With the merge rules and the application of changes, we create a new XMI file. The aim of these steps is to represent the whole information about changes and to show every problem we have found maintaining the XMI validity. First we discard the deletions and the moves which have caused violations or conflicts. Then we create the alternative mechanism to highlight conflicts and to represent possible options. Finally we insert warnings to report about everything that could cause a problem or about discarded changes. In order not to break the XMI validity, we represent alternatives and warnings with comments (like in textual merge tools) or using the XMI tag extension. This way we have a merged XMI-valid file, with all the applicable changes performed, all conflict representations and problem warnings.

On the one hand we can provide an XMI valid result, while on the other we cannot guarantee a valid model as a result. In fact, even if we are provided with three valid models (represented by XMI files), we cannot apply changes and discard them considering the correct result with respect to the model semantic, since we do not know enough about it. As an example, consider two changes that modify the minimum and the maximum of the cardinality of a relationship: we have no possibility to know if the minimum is higher than the maximum after the application of these changes, because we do not know such meanings and, consequently, we cannot avoid the occurrence of a model violation.

The approach of the merge is batch oriented, since we do not expect the developer to choose interactively from various options, but we provide a merge that represents rather than resolves problems (like conflicts, violations, etc.). In fact, the small amount of information that we could extract from XMI, permitted us to recognize changes but not to interpret them, except for low level conflicts. The batch result could be regarded as an intermediate step in the whole merge process (completed by the developer elaboration or by running another interpretation/resolution tool over it), but could be useful by itself as explained in the paper [8]. In fact, it could improve communication between parallel-working developers to resolve merge issues, or it may help developers to get a clear picture quickly about the problems concerning their work together with the others', without the necessity of finding an immediate solution (virtual merge). A clear example of how it could work is given by our tool (XMIMerge), which performs a virtual merge between the artifacts serialized by the editor ArgoUML.

6.2 Related works

In this work we proposed a “low level” merge based on the standard serialization language XMI, in which we do not have to rely on further information provided by a specific editor or by a higher level language. We have not found a similar work that deals with the merging process at XMI level and with a batch approach. However, there are some related works which are similar for some aspects, but they usually used different approaches.

We have explained (previously in this thesis) that we used a state-based approach to perform our merge. There is, however, another way to produce a merge, that is called operation-based [11]. In this approach we are provided with two sequences of changes (or operations) and the goal is to merge them. This is often put in contrast with the state-based approach. Both methods have their pros and cons, and often these depend on which method is used: for example, in the context of a state-based merge, if we have to match elements using a similarity-based algorithm, as we saw in §3.1.1, the task could be very expensive, while using identifiers is very easy and cheap. This means that in the former case we have a whole expansive merge process, while in the latter we do not. Thus we cannot simply assert that the state-based approach is more expensive. In the case of the operation-based approach, we know from somewhere (often recorded by a model editor) which operations have taken place, while in the state-based one we have to deduce them: thus, it seems better to know operations instead of deducing them. In fact, with the former approach we avoid some false positives and false negatives (sometimes we could deduce a single change from a modified element, while it could be the result of a set of operations), so if there are less problems, the developer does not have to deal with them. Nevertheless, we need a way to store operations during the modification of the artifact: it is usually carried out by the editor while the developer performs such operations. However, we choose XMI to be independent from editors: such a feature is quite valuable, since it means that this work does not rely on a precise tool or a model specification version, but it could be used in a wider setting (even though we need XMI to be more homogeneous). Thus, we have used XMI, but its main drawback is that we could not obtain information about operations: consequently, we were forced to choose the state-based approach.

Another related work is the one proposed by Westfechtel [17], in which he adopts a formal approach to provide a state-based 3-way merge of models. The fact that he proves a completely valid merged model, including moves and recognizing both context-free and context-sensible conflicts, makes his work very interesting, but still it is not suitable for our purpose. Unfortunately, we cannot apply the logic he uses to the information provided by XMI because there is no correspondence between them. In fact, as we can deduce from the title, “A Formal Approach to Three-Way Merging of EMF Models” it is based on the EMF metamodel. Since it is a new work, it may be adapted to the domain of XMI by further research.

Lindholm presented a 3-way merge on XML documents in his master thesis [12]. Since XMI is an XML dialect, the approach is very close to this work. However, considering the XMI syntax, we can deduce more information from its structure and from the serialization patterns used (that we have knowledge about), so we can make more assumptions than in a generic XML file. For this reason, even though there might have been some similar cases to analyze (since XMI files are also XML files), we provided different solutions to handle certain changes. Furthermore, we can exclude some cases that we know we cannot find in XMI; and, on the other hand, we can add some specific cases regarding only XMI. Using Ids (from the XMI specification) allows us to avoid the match part of his merge, which is the most expensive and failure prone phase. Furthermore, Ids prevent the copy operation considered in his work. Moreover, we do not need child-node order: in Lindholm's work, in fact, a change could affect a node if it is swapped with another sibling one, whilst for our purpose it does not cause relevant changes (it is the same case of an attribute placed before or after another one). Furthermore, we changed the context definition in our work. Lindholm assumes that there is a context problem when there are changes between a node and other nodes surrounding it, so every structural change on close nodes is discarded in order not to interfere with semantic (unknown) dependencies. However, we do not need to discard them: we do not know the exact context and it is not necessary to apply automatically every change (since we insert warnings and alternatives among unsolvable conflicts), so our approach highlights hypothetical problems due to proximity but without discarding changes. We use the same strategy to handle conflicts: this is one of the most important difference between the two works. In our solution, in the case of unsolvable conflicts, we include different options on the merge result which will be left to be checked by the developer. In Lindholm's work (as well as in Asklund's [2]), conflicts caused by the modification of the same property are solved choosing the “first” change. This depends, however, on the order in which we read changed versions, which breaks our requirement of symmetry (§3.1.5). Again, in the case of deletion-conflict (when we have another change on a deleted sub-tree), the deletion is performed erasing other simultaneous changes on the same deleted sub-tree. This results in a loss of information in the merged file, which we managed to avoid. On the other hand, the drawback of our approach is having a non complete merge which has to be validated again, whereas Lindholm needs to perform a merge and has to take all the decisions about all conflicts.

The last issue we discuss represents an important difference between our work and that of others described in the previous paragraphs. By choosing the batch approach and creating a merge with all the information but without all the solutions, our result presents no “dangerous” change-applications (the ones that could lead to a loss of information). It proposes a non complete merge (conflicts still need to be resolved), which means that the result needs to be elaborated again, before being considered completely merged. When dealing with models this solution seems reasonable, but if we have to merge files quickly between mobile phones (like in a scenario involving XML proposed by Lindholm), it might be preferable having a valid (but possibly not correct) merge despite some loss of information.

6.3 Further research

Our work could prove useful also for presenting further research proposals concerning the XMI approach.

First of all, XMI itself and its implementation could be improved to obtain greater homogeneity and only then to be used to perform a generic model merge. The language presents valuable characteristics such as the Id mechanism to match files and the extension tags to report annotations of different tools. However, tool vendors should provide a more standard compliant implementation in order to apply what is proposed by the specification in real life. For example the habit of using Ids and keeping them over saves and loads could be a very nice feature in order to avoid the dangerous and expensive task of matching.

Still, the language itself presents some rules of serialization that, although allows flexibility, it could lead to ambiguities and issues concerning the merge problem. One of them is the choice of nesting MOF entities as sub-structures of other nodes which represent other entities: it does not add more expressivity since, as we saw in chapter 2, the two patterns of nesting and using references are equivalent.

Moreover, XMI could be provided with a new mechanism(s) to represent warnings and alternatives. This would be very useful to represent such elements in a standard way: having a specification to followed lets everyone free to create new tools to elaborate such information derived from a merge result, without creating all the other merge steps. For this purpose, in §3.2.5 we listed some requirements to be respected in case one decides to implement the useful mechanisms of alternative and warning.

The same thing could be expressed by the metamodel: for example, the same mechanism could be described in the MOF specification, and it would have the same meaning (since XMI uses the MOF specification).

There are several ways to improve this work: first of all, the techniques described should be verified on a wide set of models which we could not perform due to a lack of time and, as said, of homogeneity in XMI artifacts. In fact, to do that, we would need a more evolute state of XMI, in which tool vendors produce more homogeneous artifacts. At that point it may be necessary to adjust the presented algorithm (in case XMI was changed).

Using the serialization pattern without nested entities allows us to have all MOF classifiers well defined and separated in different sub-trees of the root: this information may be used to create a tool which reasons over the connections among entities. We know that those connections are represented only by references. A changed reference means that a dependency between entities has been changed (for example the fact that A was included in B and now it is not), but they have not been structurally modified (a good example could be a developer performing a refactoring). Therefore, a tool could create a reference graph to study only dependencies between entities and it could try to resolve only problems concerning references.

We proposed a batch merge, a result with a lot of annotations: XMI is not very human-readable, it is a mechanism to serialize models. For this reason, a visualization tool could be very useful in order to show the developer a more userfriendly representation of the different and connected alternatives, warnings and changes, possibly with a model representation. Besides, a tool which helps resolving conflicts, violations and other context related problems could be very useful. We provided a first Java implementation of such a tool (XMIMerge), which works for the artifacts serialized by the editor ArgoUML. Some examples could be found in Appendix A and in chapter 5.

Chapter 7 Conclusion

The study of the XMI language has highlighted some problems, first of all, the fact that the same model could be represented by different XMI productions (different patterns of serialization and different versions), which means that we cannot compare a set of any XMI files. Furthermore, some patterns contain more model semantic information than others, or they represent it differently, which means that we can make assumptions when dealing with a certain pattern, but these are not valid when dealing with another one. These considerations forced us to define some restrictions which have to be satisfied in order that this work can be considered valid.

We proposed a 5-step merge process which takes as input three XMI files and provides as output a new XMI valid file that represents the merged XMI tree. Such a process makes a diff of the files relying on unique identifiers (avoiding the expensive job of matching).

The proposed algorithm, once all changes are obtained, finds conflicts among them. The algorithm works deeply, which means that two changes are in conflict only if they involve the same smallest structural element (fine-grain unit of comparison) and widely, which means that it should find every direct conflict. The algorithm warns also about syntax violations and distance-1 hypothetical non-direct conflicts; it could be extended in a way that it will be able to warn also a distance-n non-direct conflict. The algorithm merges non-conflict changes, which means that if two changes are not in direct conflict, they are applied correctly with respect to the XMI syntax.

The output is represented as a file in which we can find traces about all changes (also those in conflict), so the algorithm runs in batch-mode. In fact, it reports about all ordinary changes, unsolvable conflicts, syntax violations and hypothetical problems. To handle such reports, three approaches are used: annotations to label a changed element, alternatives which show all possible solutions for a conflict, and warnings that report about violations or context-related problems. Since such mechanisms are not supported by the XMI language, we propose the use of the extension XMI tag and XMI comments. We also provided a specification which could be verified and extended by further research.

Discussing the outputs of the algorithm (which should be verified on a wider set of examples), we can observe how this XMI approach suffers from the lack of semantic information, which leads to a lack of warranty about correct model semantic output.

Finally, at present the algorithm is not completely environment-independent, since it needs to analyze a set of XMI files which are necessarily serialized using the same pattern, and it works better on a specific pattern. However, once the XMI language found homogeneity in the specification and in the implementation performed by tool vendors, we showed how we can provide, without any further information (other than the three provided files), a preliminary XMI valid merge which includes all information about changes, conflicts and some model-semantic problems which could be elaborated subsequently by the developer or by further tools.

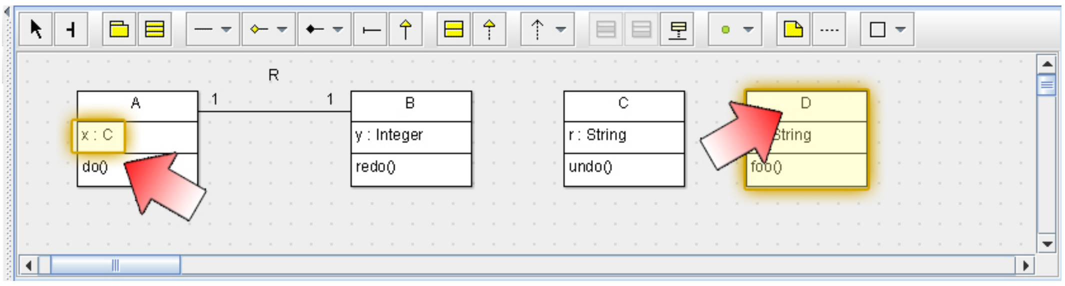
Following all these considerations, we have created a Java tool (XMIMerge) which provides a graphical virtual merge of three XMI files given as input. The tool is meant to work with artifacts serialized by the model editor ArgoUML. This strengthened our beliefs that the proposed merge process can be implemented, but it also became obvious that an XMI-based merge tool cannot be fully environment-independent at the moment.

Appendix A
Tool Examples

In this appendix we will present some examples: for each of them the changes at the model level (ArgoUML screenshots of the `CA`, `V1` and `V2`) are shown and then we can see how the tool handles the merge problems detected. In the first example `XMIMerge` reports a violation, in the second a context issue and in the third it manages a combination of two problems.

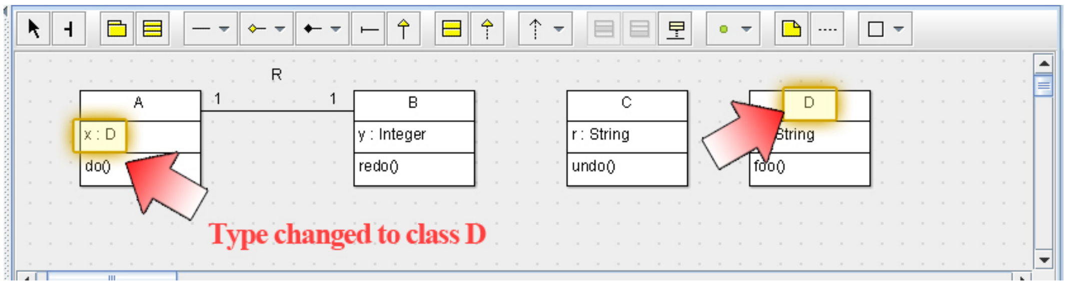
A.1 Violation example

In this example we show how the tool `XMIMerge` reports an XMI violation.

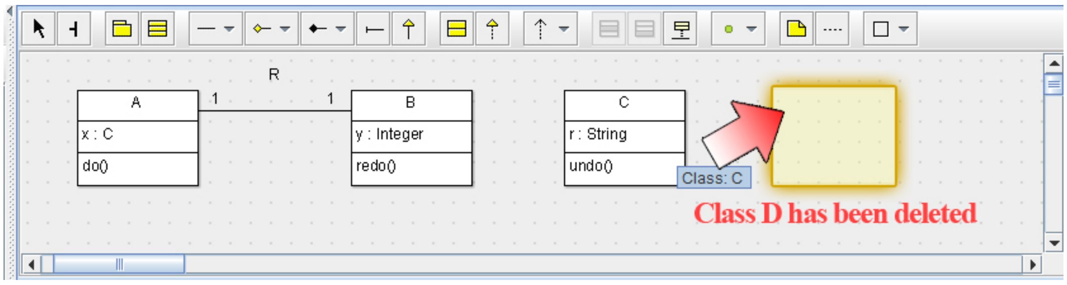


In `CA`, observe the attribute's type `C` and the class `D`.

In `V1` the type of `x` is changed into `D`:

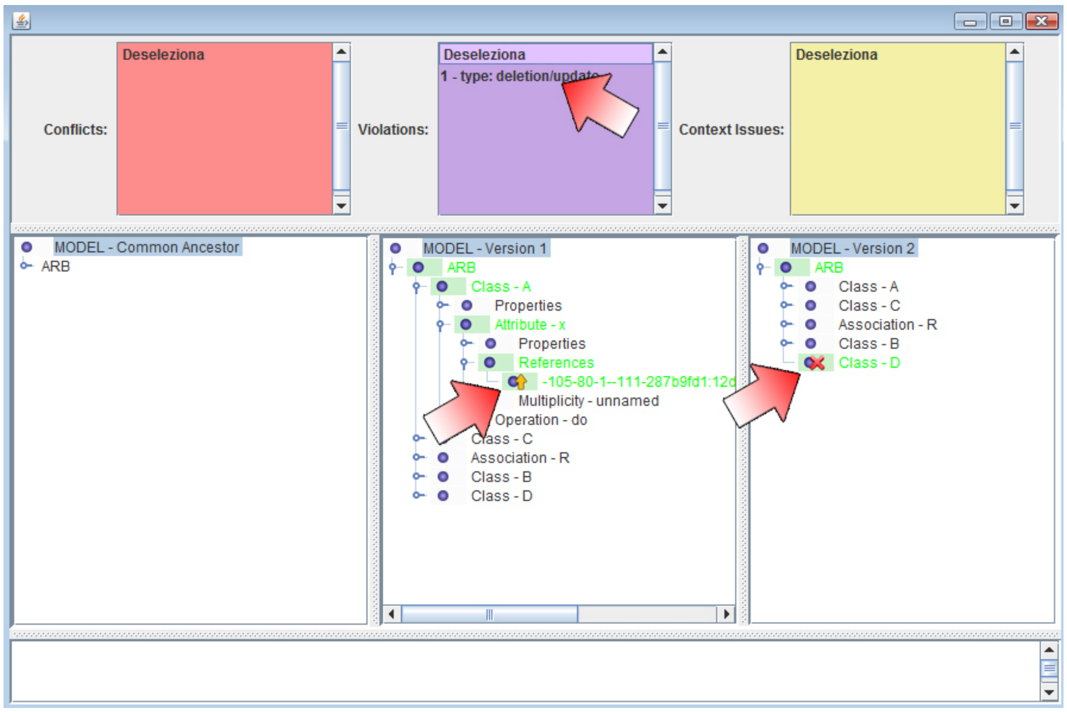


In `V2` the class `D` has been deleted:

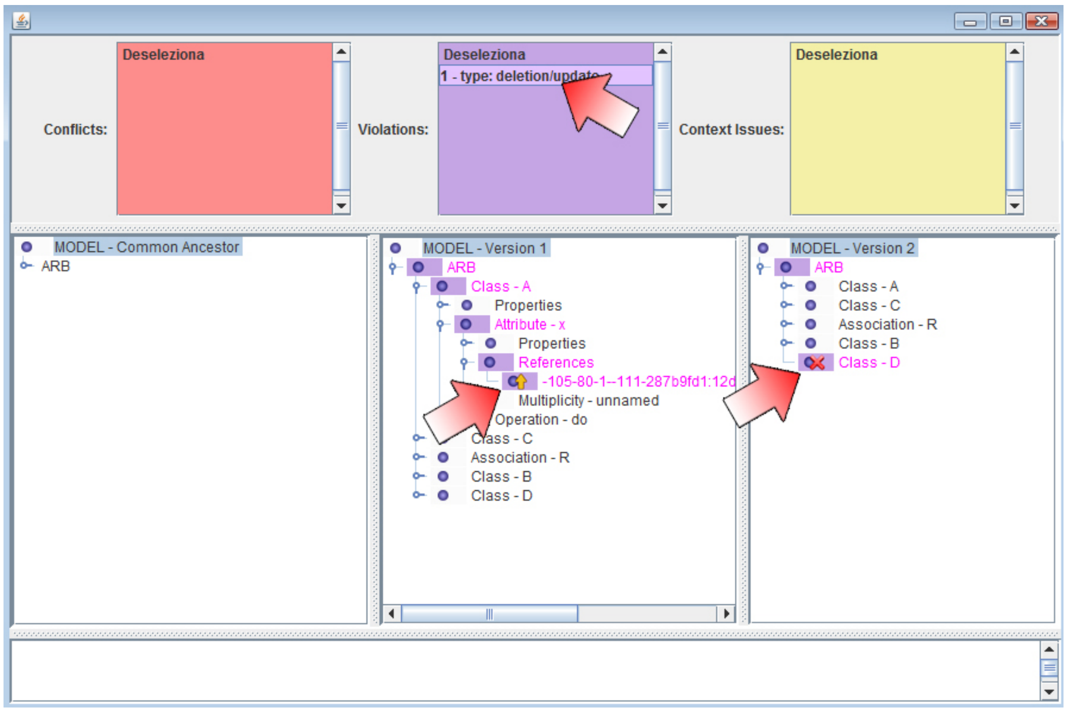


Merging `V1` and `V2` would result in a reference pointing to a non-existent element. How does `XMIMerge` handle this problem?

We can see how `XMIMerge` finds the changes:



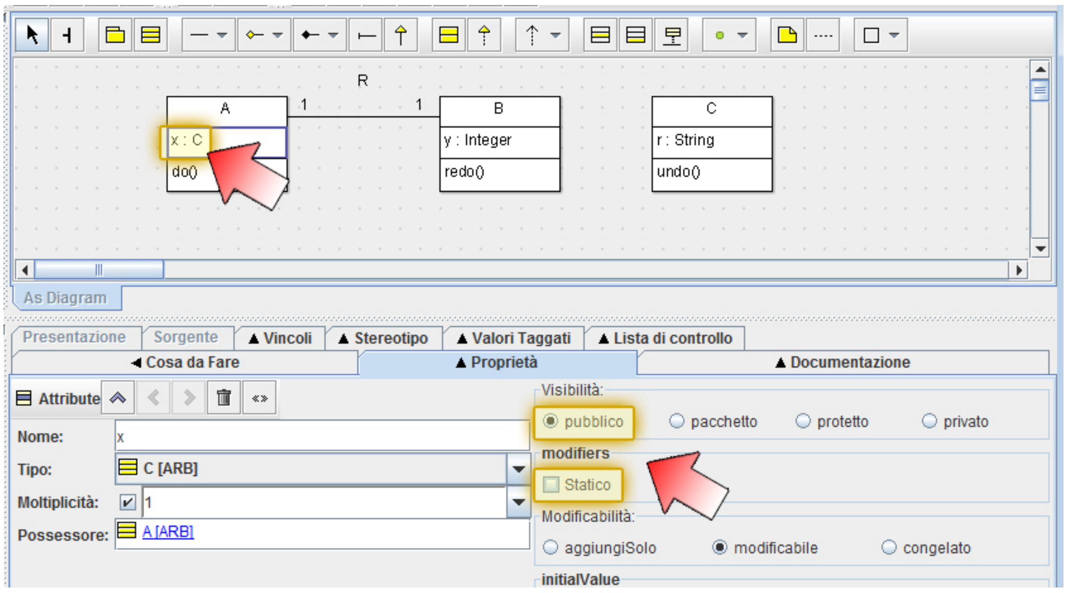
And then that it finds the violation:



A.2 Context-issue example

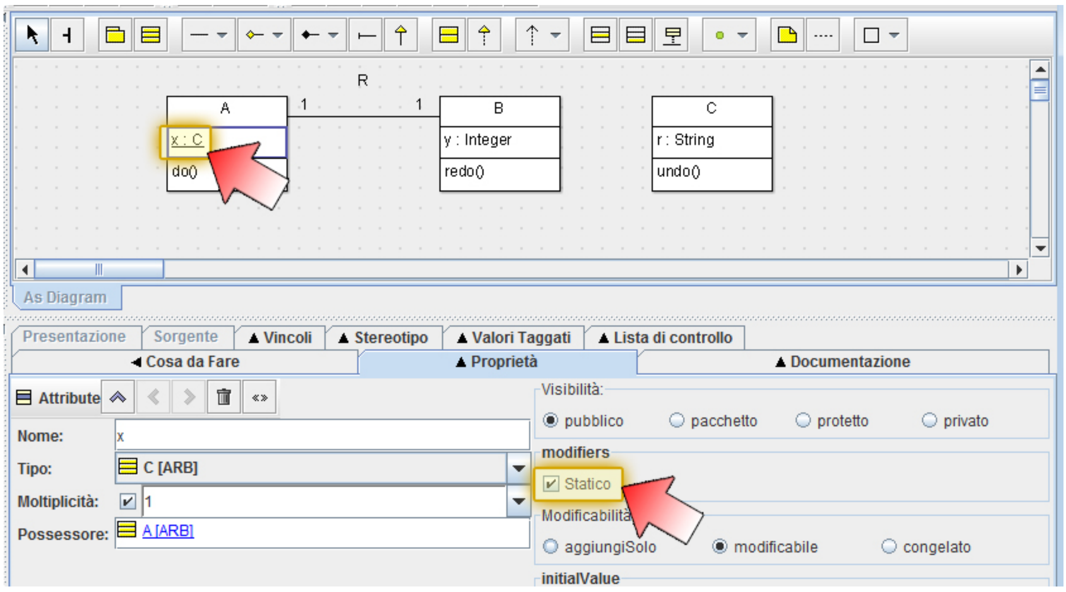
In this example we show how the tool XMIMerge reports a context issue.

In the CA note the attribute x, it is public and not static:



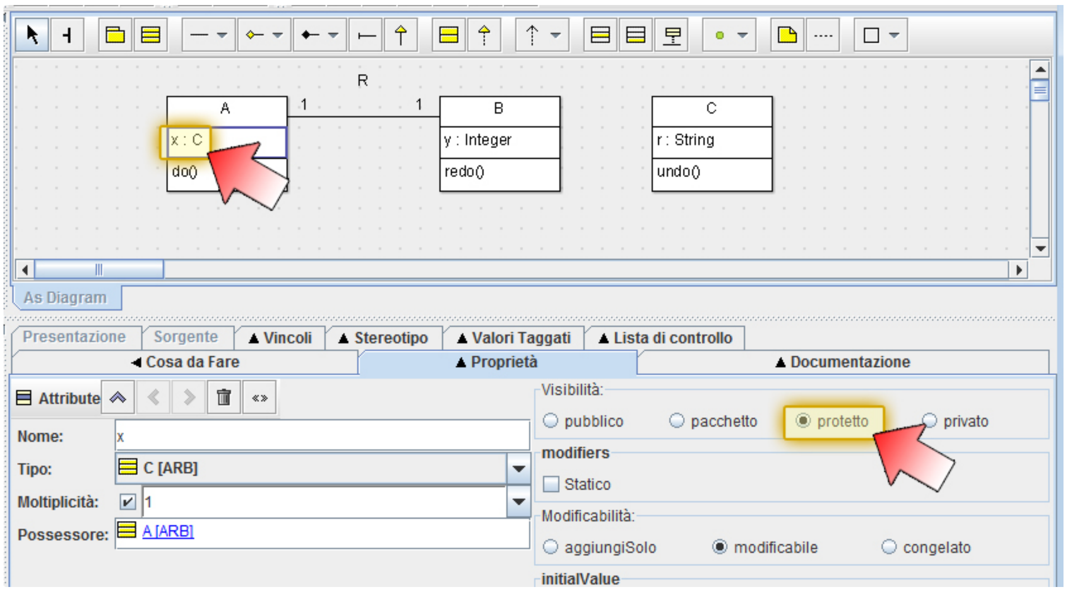
Nome: x
Tipo: C [ARB]
Molteplicità: ✓ 1
Possessore: A [ARB]
Visibilità: [✓ pubblico] pacchetto protetto privato
modifiers: ☐ Statico
Modificabilità: aggiungiSolo [✓ modificabile] congelato

In V1 the attribute x has been changed and now it is static:



Nome: x
Tipo: C [ARB]
Molteplicità: ✓ 1
Possessore: A [ARB]
Visibilità: [✓ pubblico] pacchetto protetto privato
modifiers: ✓ Statico
Modificabilità: aggiungiSolo [✓ modificabile] congelato

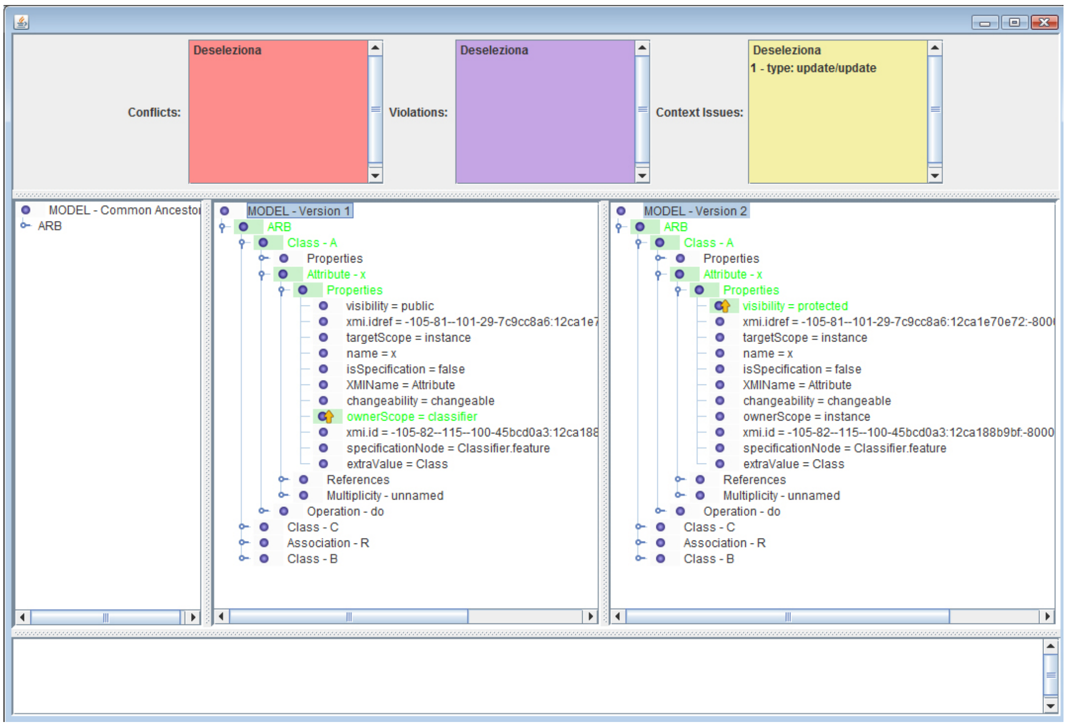
In `V2`, `x` is set to `protected`:



Nome: x
Tipo: C [ARB]
Moltiplicità: 1
Possessore: A [ARB]

Visibilità: pubblico pacchetto [✓ protetto] privato
modifiers: Statico
Modificabilità: aggiungiSolo [✓ modificabile] congelato
initialValue

We are not sure whether these two changes could cause a problem or not. However, `XMIMerge` finds two changes on the same element, and it reports a context issue with low priority (for example, `30`):



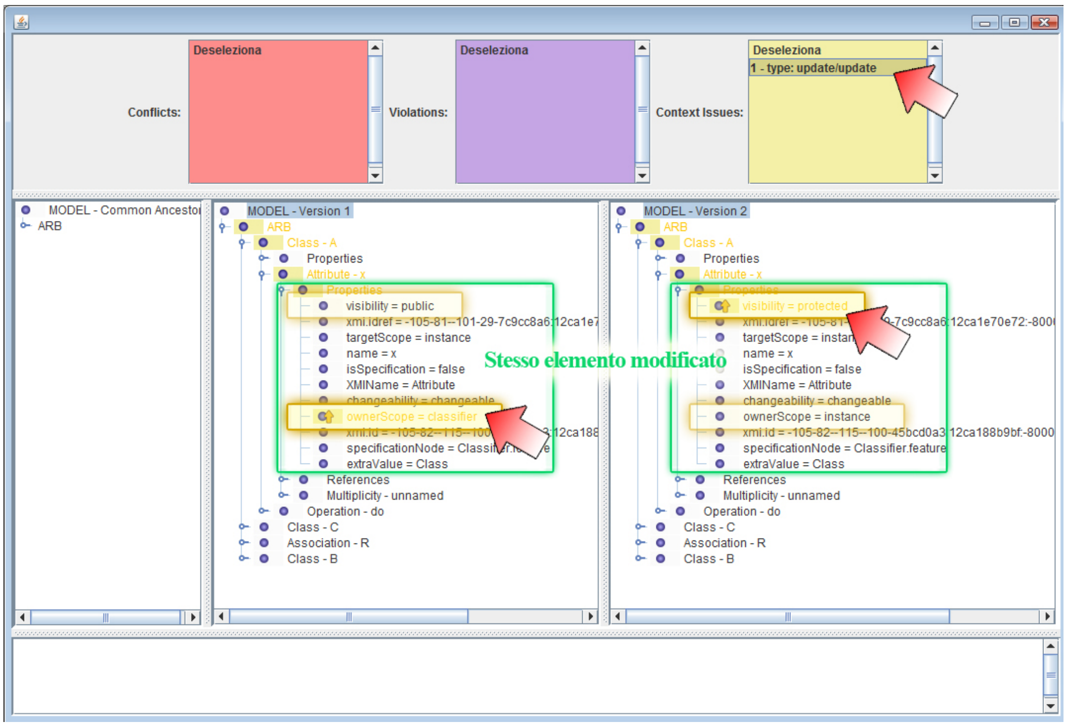
Conflicts:
Violations:
Context Issues: 1 - type: update/update

```
MODEL - Common Ancestor
ARB
```

```
MODEL - Version 1
ARB
  Class - A
    Properties
    Attribute - x
      Properties
        visibility = public
        xmi.idref = -105-81--101-29-7c9cc8a6:12ca1e7
        targetScope = instance
        name = x
        isSpecification = false
        XMIName = Attribute
        changeability = changeable
        ownerScope = classifier
        xmi.id = -105-82--115--100-45bcd0a3:12ca188
        specificationNode = Classifier.feature
        extraValue = Class
      References
      Multiplicity - unnamed
    Operation - do
  Class - C
  Association - R
  Class - B
```

```
MODEL - Version 2
ARB
  Class - A
    Properties
    Attribute - x
      Properties
        visibility = protected
        xmi.idref = -105-81--101-29-7c9cc8a6:12ca1e70e72:-800
        targetScope = instance
        name = x
        isSpecification = false
        XMIName = Attribute
        changeability = changeable
        ownerScope = instance
        xmi.id = -105-82--115--100-45bcd0a3:12ca188b9bf:-8000
        specificationNode = Classifier.feature
        extraValue = Class
      References
      Multiplicity - unnamed
    Operation - do
  Class - C
  Association - R
  Class - B
```

If we select the reported issue the tool shows where the problem could be:



```
MODEL - Common Ancestor
ARB
```

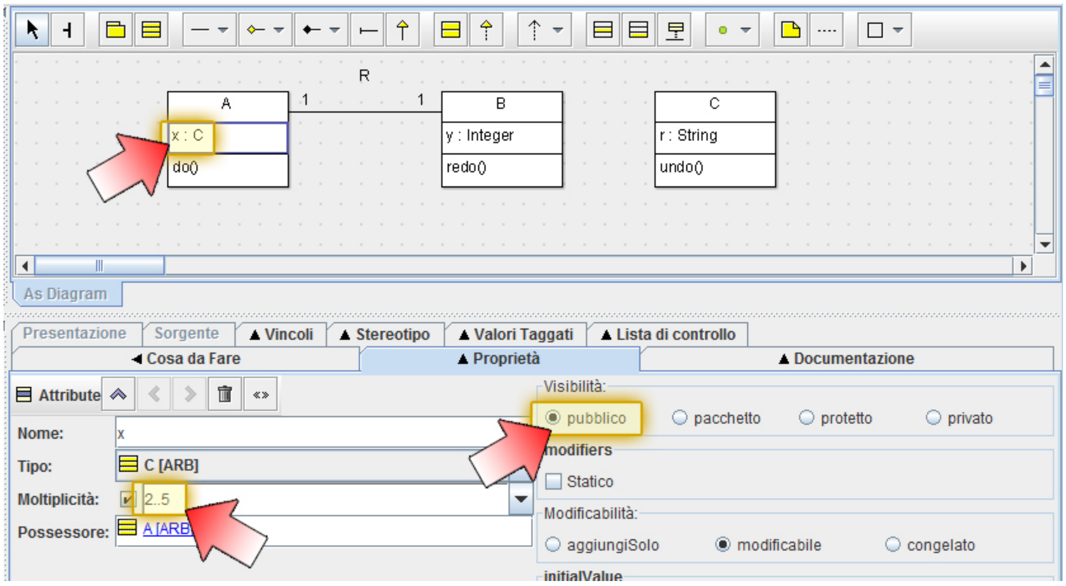
```
MODEL - Version 1
ARB
  Class - A
    Properties
    Attribute - x
      Properties
        visibility = public
        xmi.idref = -105-81--101-29-7c9cc8a6:12ca1e7
        targetScope = instance
        name = x
        isSpecification = false
        XMIName = Attribute
        changeability = changeable
        ownerScope = classifier
        xmi.id = -105-82--115--100
        specificationNode = Classifier.feature
        extraValue = Class
      References
      Multiplicity - unnamed
    Operation - do
  Class - C
  Association - R
  Class - B
```

```
MODEL - Version 2
ARB
  Class - A
    Properties
    Attribute - x
      Properties
        visibility = protected
        xmi.idref = -105-81-  -7c9cc8a6:12ca1e70e72:-800
        targetScope = instance
        name = x
        isSpecification = false
        XMIName = Attribute
        changeability = changeable
        ownerScope = instance
        xmi.id = -105-82--115--100-45bcd0a3:12ca188b9bf:-8000
        specificationNode = Classifier.feature
        extraValue = Class
      References
      Multiplicity - unnamed
    Operation - do
  Class - C
  Association - R
  Class - B
```

A.3 Combination example

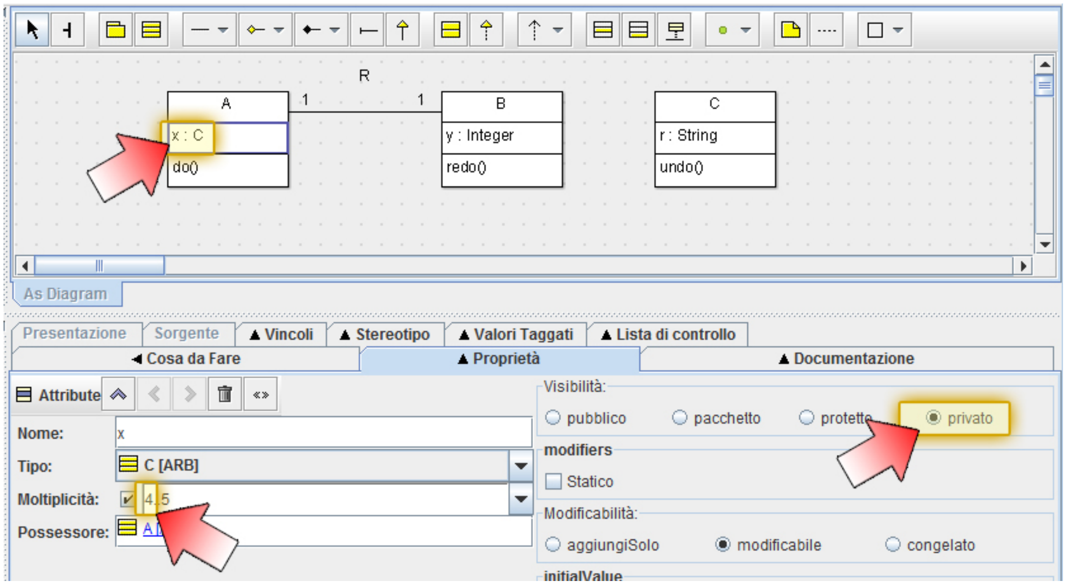
In this example we can see how `XMIMerge` handles multiple problems.

In the `CA` pay attention on the cardinality of `x` (`Molteplicità`) which says that this attribute can occur at least `2` and at most `5` times. Moreover, let's see also its `visibility` (`Visibilità`) which is set to `public`:



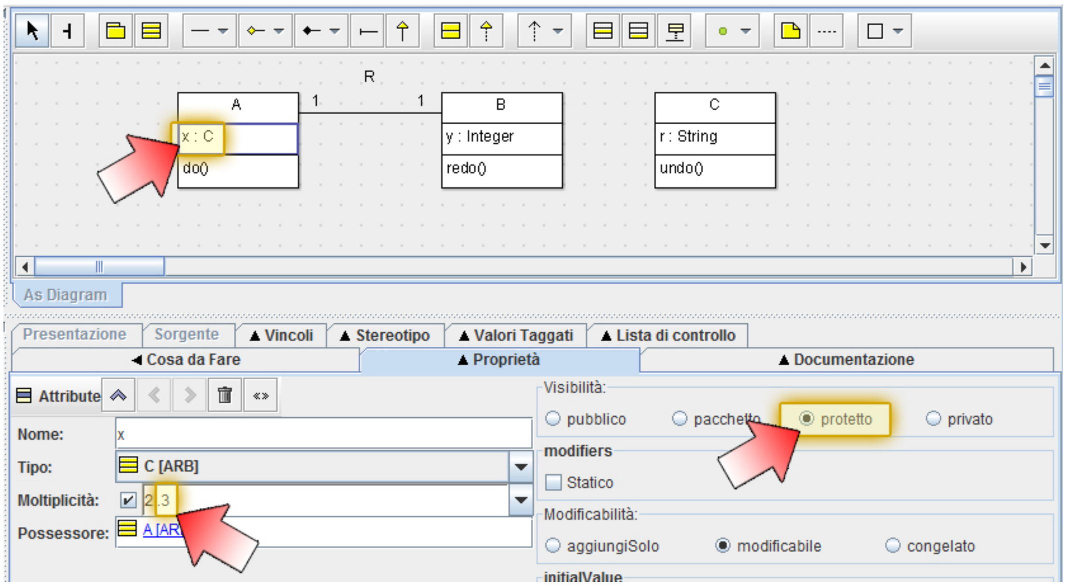
Nome: x
Tipo: C [ARB]
Molteplicità: 2..5
Possessore: A [ARB]
Visibilità: [✓ pubblico] pacchetto protetto privato
modifiers: Statico
Modificabilità: aggiungiSolo [✓ modificabile] congelato

In `V1` the lower bound of the cardinality of `x` has been incremented to `4`, while the `Visibility` is now `private`:



Nome: x
Tipo: C [ARB]
Molteplicità: 4..5
Possessore: A
Visibilità: pubblico pacchetto protetto [✓ privato]
modifiers: Statico
Modificabilità: aggiungiSolo [✓ modificabile] congelato

In `V2` the upper bound has been decremented to `3` and the `Visibility` has changed to be `protected`:



Nome: x

Tipo: C [ARB]

Moltiplicità: ✓ 2..3

Possessore: A [AR

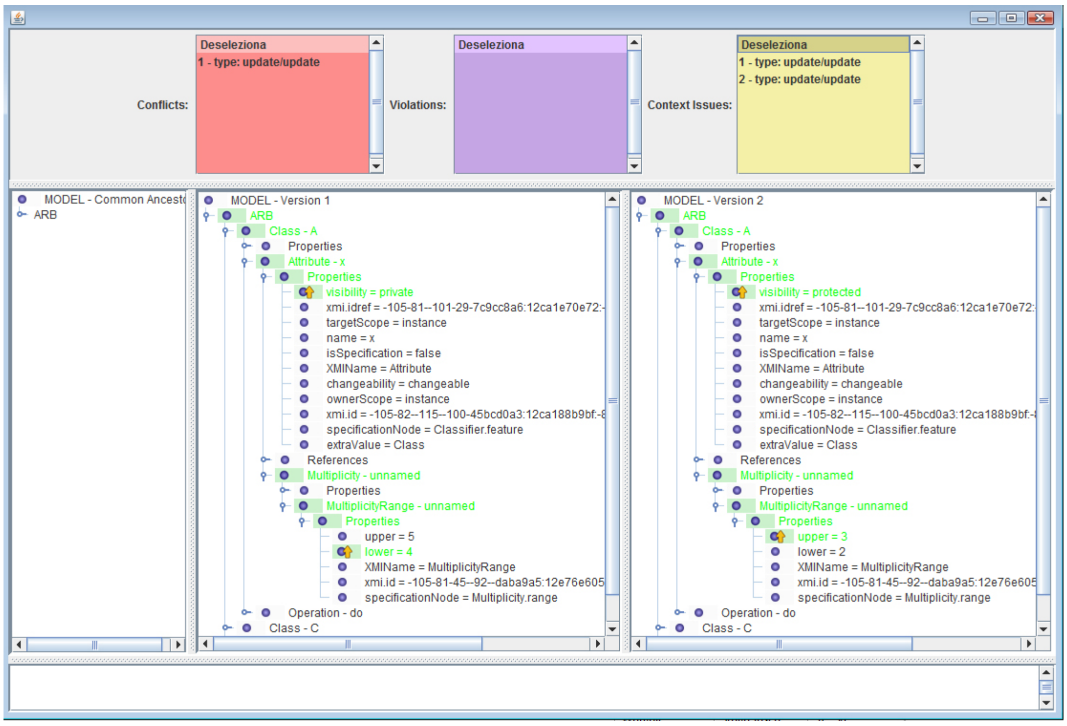
Visibilità: pubblico pacchetto [✓ protetto] privato

modifiers: Statico

Modificabilità: aggiungiSolo [✓ modificabile] congelato

initialValue

Here we have two merge problems: the first is that merging the cardinality we will have the lower bound set to `4` and the upper bound set to `3`, which is clearly a UML problem thus the model is invalid. Moreover, we cannot set the `Visibility` of the attribute to `protected` and `private` at the same time. `XMIMerge` detects the changes (green), a conflict and two context issues:



Conflicts: Deseleziona
1 - type: update/update

Violations: Deseleziona

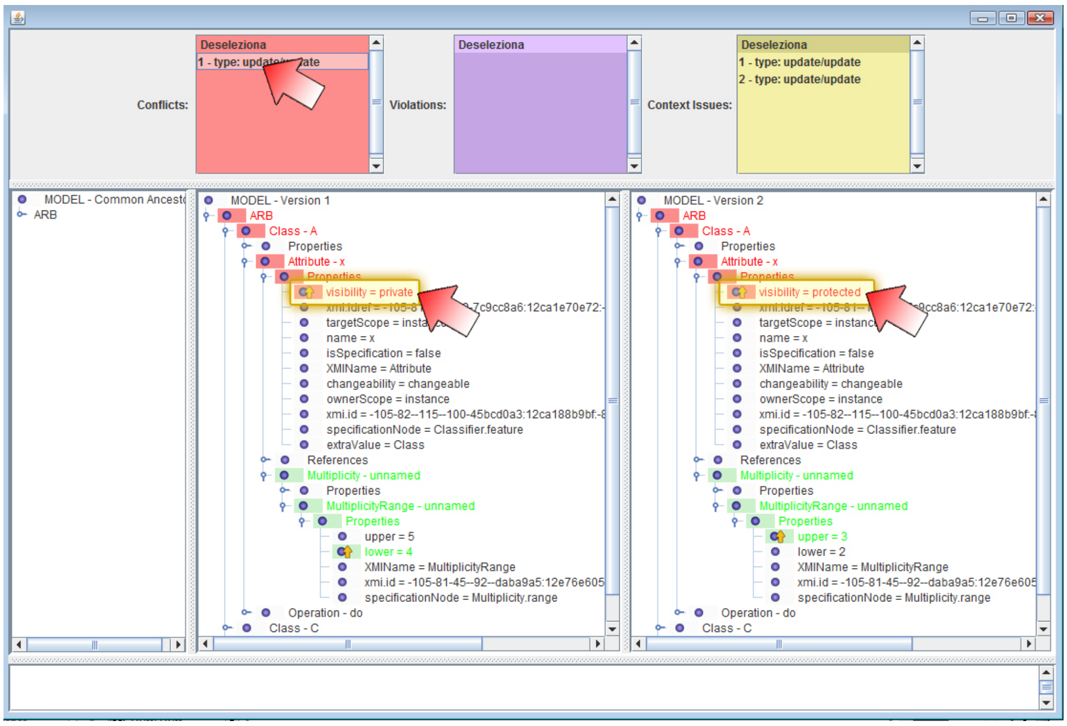
Context Issues: Deseleziona
1 - type: update/update
2 - type: update/update

```
MODEL - Common Ancesto
ARB
```

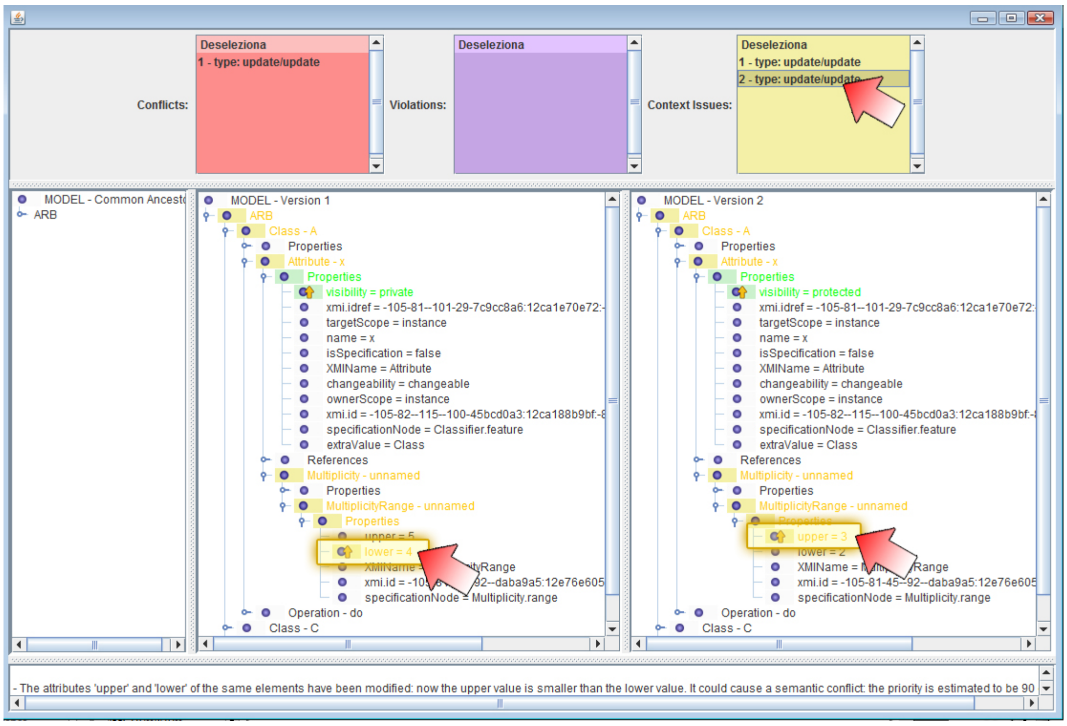
```
MODEL - Version 1
ARB
  Class - A
    Properties
    Attribute - x
      Properties
        visibility = private
        xmi.idref = -105-81--101-29-7c9cc8a6:12ca1e70e72:-
        targetScope = instance
        name = x
        isSpecification = false
        XMIName = Attribute
        changeability = changeable
        ownerScope = instance
        xmi.id = -105-82--115--100-45bcd0a3:12ca188b9bf:-8
        specificationNode = Classifier.feature
        extraValue = Class
      References
      Multiplicity - unnamed
        Properties
        MultiplicityRange - unnamed
          Properties
            upper = 5
            lower = 4
            XMIName = MultiplicityRange
            xmi.id = -105-81-45--92--daba9a5:12e76e605
            specificationNode = Multiplicity.range
    Operation - do
  Class - C
```

```
MODEL - Version 2
ARB
  Class - A
    Properties
    Attribute - x
      Properties
        visibility = protected
        xmi.idref = -105-81--101-29-7c9cc8a6:12ca1e70e72:-
        targetScope = instance
        name = x
        isSpecification = false
        XMIName = Attribute
        changeability = changeable
        ownerScope = instance
        xmi.id = -105-82--115--100-45bcd0a3:12ca188b9bf:-8
        specificationNode = Classifier.feature
        extraValue = Class
      References
      Multiplicity - unnamed
        Properties
        MultiplicityRange - unnamed
          Properties
            upper = 3
            lower = 2
            XMIName = MultiplicityRange
            xmi.id = -105-81-45--92--daba9a5:12e76e605
            specificationNode = Multiplicity.range
    Operation - do
  Class - C
```

We can select the conflict from the list to display only that specific problem. `XMIMerge` found the `Visibility` conflict. The other changes remain green:



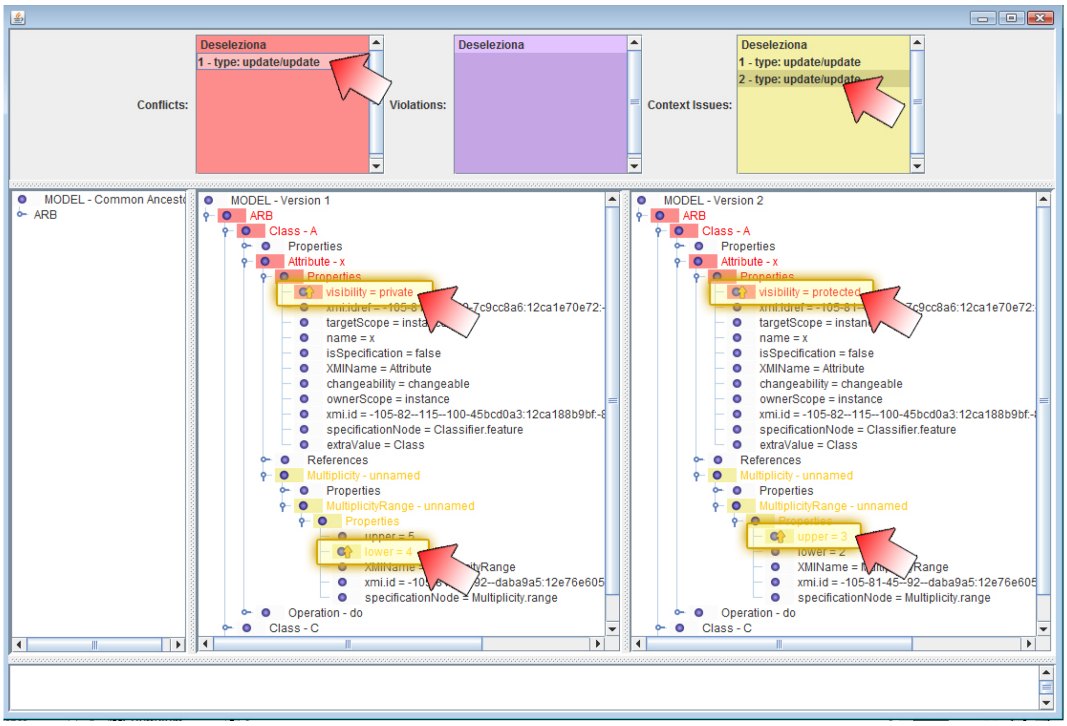
We can then select one of the context issues:



- The attributes 'upper' and 'lower' of the same elements have been modified: now the upper value is smaller than the lower value. It could cause a semantic conflict: the priority is estimated to be 90

There appear two context issues, but they are, as a matter of fact, the same issue which has been considered separately by two different detections. One with low priority (`30` out of `100`) reports that the same element has been changed, whilst the other one specifically finds the upper – lower problem, which is registered with a value of `90` out of `100` (higher priority).

Now we can select both the displayed problems: the tool shows both of them (the common path has been colored with the color of the last problem selected):



```
1 - type: update/update
```

```
1 - type: update/update
2 - type: update/update
```

```
MODEL - Common Ancesto
ARB
```

```
MODEL - Version 1
ARB
  Class - A
    Properties
    Attribute - x
      Properties
        visibility = private
        xmi.idref = -105-8           9-7c9cc8a6:12ca1e70e72:-
        targetScope = insta
        name = x
        isSpecification = false
        XMIName = Attribute
        changeability = changeable
        ownerScope = instance
        xmi.id = -105-82--115--100-45bcd0a3:12ca188b9bf:-8
        specificationNode = Classifier.feature
        extraValue = Class
      References
      Multiplicity - unnamed
        Properties
        MultiplicityRange - unnamed
          Properties
            upper = 5
            lower = 4
            XMIName =          ityRange
            xmi.id = -105        92--daba9a5:12e76e605
            specificationNode = Multiplicity.range
    Operation - do
  Class - C
```

```
MODEL - Version 2
ARB
  Class - A
    Properties
    Attribute - x
      Properties
        visibility = protected
        xmi.idref = -105-81-          7c9cc8a6:12ca1e70e72:
        targetScope = instan
        name = x
        isSpecification = false
        XMIName = Attribute
        changeability = changeable
        ownerScope = instance
        xmi.id = -105-82--115--100-45bcd0a3:12ca188b9bf:-
        specificationNode = Classifier.feature
        extraValue = Class
      References
      Multiplicity - unnamed
        Properties
        MultiplicityRange - unnamed
          Properties
            upper = 3
            lower = 2
            XMIName = M        Range
            xmi.id = -105-81-45--92--daba9a5:12e76e605
            specificationNode = Multiplicity.range
    Operation - do
  Class - C
```

Bibliography

[1] ArgoUML open source project, argouml.tigris.org.

[2] Asklund, U., *Identifying conflicts during structural merge,* Proceedings of Nordic Workshop on Programming Environment Research, 1994.

[3] Babich, W. A., *Software Configuration Management – Coordination for Team Productivity*, Addison-Wesley, 1986

[4] Baisley, D. E., *Method in a computer system for comparing XMI-based XML document for identical contents*, 1999

[5] Bendix, L.; Emanuelsson, P.: *Collaborative Work with Software Models - Industrial Experience and Requirements*; p.x in: Proc. 2nd Intl. Conf. Model Based Systems Engineering - MBSE'09, Haifa, Israel, March 2-6, 2009; http://www.mbse-org.org/; 2009

[6] Bendix, L.; Emanuelsson, P.: *Diff And Merge Support For Feature Oriented Development*; p.31-34 in: Proc. 2008 ICSE Workshop on Comparison and Versioning of Software Models, May 17, 2008, Leipzig; ACM; 2008

[7] Bendix, L.; Emanuelsson, P.: *Requirements for Practical Model Merge - An Industrial Perspective*; p.167-180 in: Proc. 12th Int.l Conf Model Driven Engineering Languages and Systems, MODELS 2009, Denver, CO, USA, October 4-9, 2009; LNiCS 5795, Springer; 2009

[8] Bendix, L., Koegel, M., Martini, A., *The Case for Batch Merge of Models – Issues and Challenges – International Workshop on Models and Evolution* - ME 2010, Oslo, Norway, October 3, 2010.

[9] Fowler, M., *“Continuous Integration”*, published on the internet, URL: http://martinfowler.com/articles/continuousIntegration.html

[10] Grose, T., Doney, G., & Brodsky, S., *Mastering XMI: Java Programming with XMI, XML, and UML*, Wiley, New York, 2002.

[11] Kögel, M., Hermannsdoerfer, M., von Wesendonk, O., Helming, J., *Operation-based Conflict Detection on Models*, in proceedings of the International Workshop on Model Comparison in Practice, Malaga, Spain, July 1, 2010.

[12] Lindholm, T., Master Thesis, *A 3-way Merging Algorithm for synchronizing ordered trees - the 3DM merging and differencing tool for XML*, Helsinki University of Technology,

[13] Oliviera, H., Murta, L., Werner, C., *Odyssey-VCS: A Flexible Version Control System for UML Model Elements*, in proceedings of the 12th International Workshop on Software Configuration Management, Lisbon, Portugal, September 5-6, 2005.

[14] OMG-XML Metadata Interchange (XMI) Specification, version 1.0-2.0 http://www.omg.org/technology/documents/formal/xmi.htm

[15] Pagano, D., Brüggemann-Klein, A., *Engineering Document Applications*, From UML Models to XML Schemas. Presented at Balisage: The Markup Conference 2009, Montréal, Canada, August 11 - 14, 2009. In Proceedings of Balisage: The Markup Conference 2009. Balisage Series on Markup Technologies, vol. 3 (2009). doi:10.4242/BalisageVol3.Bruggemann-Klein01

[16] Persson, A., Gustavsson, H., Lings, B., Lundell, B., Mattsson, A., Ärlig, U., *OSS tools in a heterogeneous environment for embedded systems modelling: an analysis of adoptions of XMI*, 2005

[17] Westfechtel, B., *A Formal Approach to Three-Way Merging of EMF Models*, in proceedings of the Workshop on Model Comparison in Practice, Malaga, Spain, July 1, 2010.

[18] Wikipedia, http://en.wikipedia.org/wiki/Unified_Modeling_Language