Managing model conflicts in distributed development *

A. Cicchetti, D. Di Ruscio, and A. Pierantonio

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

Abstract. The growing complexity of current software systems naturally conveyed their development toward incremental and distributed approaches to speed up the process. Several developers update the same artefact operating concurrent manipulations which need to be coherently combined. The interaction among those changes inevitably involves conflicts which must be detected and reconciled.
This paper proposes a domain specific language able to define and manage conflicts caused by cooperative updates over the same model elements. The approach relies on a model-based representation of model differences and enables the specification and the detection of both syntactical and semantic conflicts.

1 Introduction

Software engineering projects are inherently cooperative, requiring many software engineers to coordinate their efforts to produce large systems [1]. With models becoming more and more commonplace, the collaboration among software developers in a distributed environment must increasingly consider also the management of models lifecycle [2]. For the collaboration based on software artefacts, version control systems are frequently used and play an important role among software engineers. However, the documents stored in these tools are almost code-level programs, and lack a reasonable organization and abstraction from designer's perspective. In this respect, merging documents representing modifications of models in a distributed environment is a challenging operation, both for its effectiveness and technical intricacy. In fact, every time modifications are merged, they may compete on the same resources giving place to conflict related issues. Conflicts can be distinguished into *syntactic* and *semantic* ones [3]. The former refers to modifications which interfere from a syntactic point of view, e.g. two modifiers give in parallel two different names to the same model element. The latter consists of collisions which are implicit and cannot be inferred from the structure of the performed modifications only.

A number of works have been proposed to deal with the problem of conflict management (see Sect. 4 for a discussion); nevertheless, they are usually based on implicit mechanisms for conflict detection, i.e. an *a priori* evaluation to understand which problems can arise and what are the ones to be checked. This makes impossible to find a

* Partially supported by the European Communitys 7th Framework Programme (FP7/2007-2013), grant agreement n. 214898.

technique capable of an arbitrary accuracy [3] and forces the designer to find a trade-off between *false-positive* and *false-negative* occurrences (in a similar way to e-mail spam filters).

This work proposes a conflict definition technique consisting of a domain-specific language able to specify both syntactic and semantic conflicts endowed with associated resolution criteria. To this end, *difference models* (already presented in [4]) are used to represent modifications between subsequent versions of a model. Therefore, conflicts are formalized in terms of relations between difference models representing parallel and conflicting modifications. Arbitrary scenarios can be described enabling the identifications of semantics-related patterns usually neglected by the traditional structural-based methods. The aim is to pursue a flexible conflict management toward the support of domain specific versioning through customizable conflict sets.

The paper is organized as follows: Sect. 2 introduces the difference representation approach which underpins the proposed techniques. Next section illustrates the conflict specification metamodel and its application by means of a running example. Sect. 4 gives an overview of related works and finally, Sect. 5 draws some conclusions and presents possible perspective work.

2 Background

Increasingly, complex software systems are cooperatively designed in distributed environments and suitable techniques are required to detect and represent the various design modifications software systems undergo during their life-cycle. In the same way, changes performed in parallel need to be analyzed since they may cause conflicts which require to be managed.

This section recalls the technical background which underpins the approach proposed in this paper. In particular, Sect. 2.1 outlines the technique which is adopted in this work to represent model differences. The technique relies on suitable *difference models* which can be composed both in sequence and in parallel to represent more complex modifications as described in Sect. 2.2.

2.1 Representing model differences

The problem of model differences is intrinsically complex and requires specialized algorithms and notations [5]. Encoding the relevant information about modifications as models allows the designer to derive from model differences powerful and interesting artefacts. In particular, they enable a wide range of possibilities, such as reconstructing the final model starting from an initial one, performing subsequent analysis, or detecting and manipulating conflicts. In this paper, model differences are represented according to the approach proposed in [4] and illustrated in Fig. 1: given two *base models* M_1 and M_2 which conform to an arbitrary *base metamodel MM*, their difference Δ conforms to a *difference metamodel MMD* derived from the former by means of an automated transformation `MM2MMD`. The approach does not impose any restriction over the metamodel *MM*, i.e. it is *metamodel-independent* and can be applied to any arbitrary modeling

language as the simplified UML metamodel in Fig. 2. In particular, the metamodel extension implemented in the `MM2MMD` transformation consists of adding new constructs able to represent the possible modifications that can occur on models that are *additions*, *deletions*, and *changes*. For instance, the application of the `MM2MMD` transformation to the sample UML metamodel in Fig. 2 produces the difference metamodel in Fig. 3: essentially, for each metaclass `MC` of the source metamodel, the additional metaclasses `AddedMC`, `DeletedMC`, and `ChangedMC` are produced.

The generated difference metamodel is able to represent all the differences amongst models which conform to the base metamodel. For example, the instances of the new metaclass `AddedClass` in Fig. 3 can be used to represent additions of new classes in the initial model. In a similar way, the metaclasses which extend the `Deleted` one will be used to specify deletions of existing model elements. Interestingly, the difference metamodel is able to represent also model updates by means of instances of `Changed` elements. For instance, the visibility modification of the attribute `mouse` performed on the model M_1 in Fig. 4 leading to M_2 is represented in the upper side of the delta model in Fig. 5.a by means of `ChangedClass`, `ChangedAttribute`, and `ChangeOperation` elements. `Changed` elements are also used in the difference model in Fig. 5.b to represent the modifications performed in the model M_1 in Fig. 4 to obtain M_3. In particular, a `ChangedClass` instance is defined to represent the changes affecting the class `Mouse` in which a new constructor has been added.

The difference representation mechanism introduced above satisfies a number of properties, as illustrated in [4]. One of them is the *applicability*, i.e. difference models can be exploited to re-apply changes to arbitrary input models (see [4] for further details) and for managing model co-evolution induced by metamodel manipulations [6]. Another interesting property is the *compositionality*, that is the possibility to combine difference models by means of operators like the sequential and the parallel ones. This is crucial in cooperative environments where many software engineers have to coordinate their efforts to produce large software systems [1]. In this respect, combining difference models can give place to conflict issues which need to be resolved, as discussed in the rest of the section.

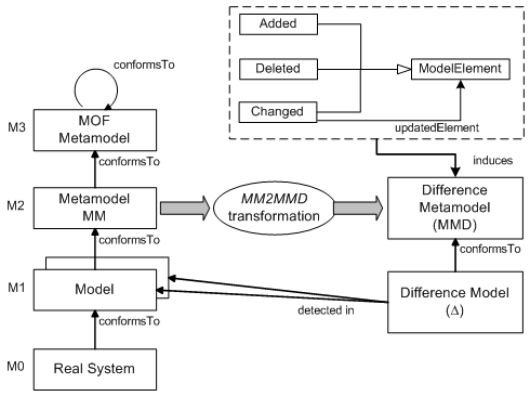

Fig. 1. Overall structure of the model difference representation approach

2.2 Composition of model differences

The evolution of a model consists of the initial specification and a number of difference models in such a way the final model can be obtained by applying all the modifications to the original one. In general, it can be convenient composing the difference models in order to obtain a unique one capturing all the occurred modifications. The composition can be performed in sequence and/or in parallel. In particular, if we consider only two subsequent modifications represented in the difference models Δ_1 and Δ_2, their sequential composition corresponds to merging the modifications conveyed by the first document with the second one, i.e.

$$\Delta = \Delta_1 \ ; \ \Delta_2 \tag{1}$$

with ";" the sequential composition operator. The resulting difference model contains a minimal difference set, i.e. only those modifications which have not been overridden by subsequent modifications. Interestingly, the representation technique has operators which are compositional, i.e. they are algebraically compatible with the induced transformations [4]. More precisely, the transformation $T_{\Delta_1 \ ; \ \Delta_2}$ induced by the composed difference model $\Delta_1 \ ; \ \Delta_2$ is completely defined by the transformations T_{Δ_1} and T_{Δ_2}, respectively, i.e.:

$$T_{\Delta_1 \ ; \ \Delta_2} = T_{\Delta_1} \cdot T_{\Delta_2} \tag{2}$$

with ";" as before and $\cdot$ an appropriate composition operator among transformation, i.e. the functional composition of transformations. Sequential compositionality is always assured by the sequential independence condition (see [3] for an extended discussion on the topic), i.e. when the modifications do not interfere with each other and can take place independently in any order.

Parallel compositions are exploited to combine modifications operated from the same ancestor in a concurrent way. In case both manipulations are not affecting the same model elements they are said to be *parallel independent* and their composition is obtained by merging the difference models. More formally, two difference models Δ_1 and Δ_2 are parallel independent if the following condition holds

$$\Delta_1 \mid \Delta_2 = (\Delta_1 \ ; \ \Delta_2) + (\Delta_2 \ ; \ \Delta_1) \tag{3}$$

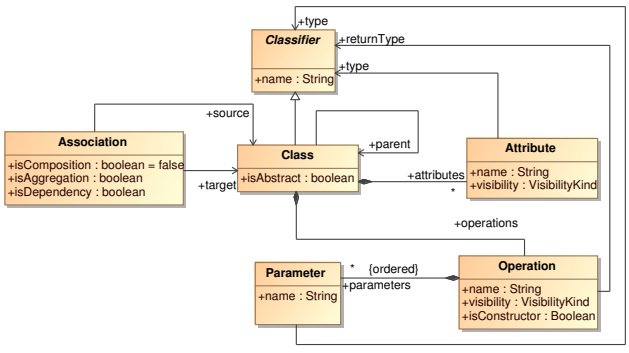

Fig. 2. Sample UML metamodel

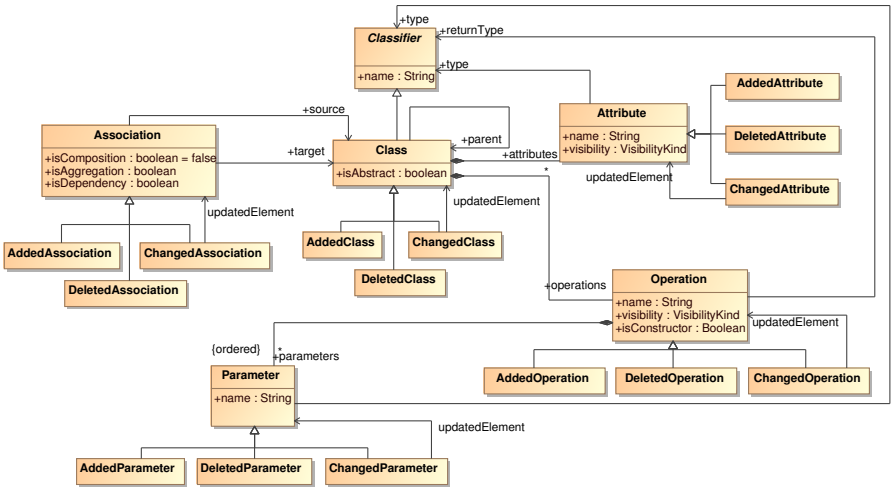

Fig. 3. Sample UML difference metamodel

where "+" denotes the non determinist choice. In essence, their application is not affected by the adopted order since they do not present any interdependencies. Otherwise, Δ_1 and Δ_2 are referred to as *parallel dependent* since conflict issues arise which need to be detected and eventually resolved. This is the case of the sample modifications $\Delta_{1,2}$, and $\Delta_{1,3}$ reported in Fig. 5. In particular, $\Delta_{1,2}$ represents the necessary modifications required to apply the *singleton* design pattern [7] (which will be considered throughout the paper as explanatory example) to the first version of the class `Mouse` reported in Fig. 4.a: the class constructor and the attribute `mouse` have been changed to private, and the new operation `getInstance()` has been added. The modifications represented in $\Delta_{1,3}$ consist of the addition of the new constructor `Mouse(posX: Integer, posY: Integer)` keeping the rest of the model unchanged (see Fig. 4.c).

According to (1) and (3), $\Delta_{1,2}$ and $\Delta_{1,3}$ in Fig. 5 are neither sequentially nor parallel independent. In fact, they interfere with the visibility of the attribute `mouse` and of the constructor `Mouse()`. In general, this kind of conflicts are called *syntactic conflict* (since the modifications interfere from a syntactical point of view) and there are a number of approaches which are able to detect them [3], even though their resolution generally requires manual interventions. For instance, according to [8] in order to maintain the singleton modification, the conflicts can be resolved by applying $\Delta_{1,3}$ and subsequently $\Delta_{1,2}$. In this way all the modifications are merged and the conflicting ones represented in $\Delta_{1,3}$ are overwritten by those in $\Delta_{1,2}$ leading to the target model in Fig. 6.

Although all the syntactic conflicts have been resolved, the obtained model in Fig. 6 does not adhere to the singleton design pattern prescriptions. In fact, taking into account the semantics behind the singleton, the accessibility of the new constructor `Mouse(posX: Integer, posY: Integer)` breaks that design pattern principles. This is due to parallel changes which interfere on a level which concerns the underlying pattern semantics

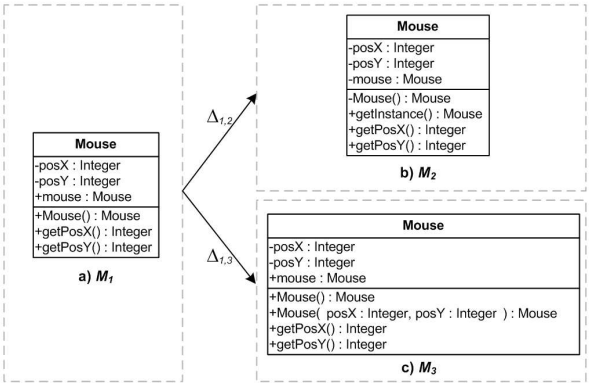

Fig. 4. Sample model modifications

and that are not syntactically detectable. In particular, the addition of the new constructor specified in the model $\Delta_{1,3}$ does not conflict with any modification represented in $\Delta_{1,2}$, even though it raises problems with respect to the semantic of the singleton design pattern. In general, this kind of conflicts are called *semantic conflicts* (according to the terminology in [3]) and demand explicit techniques to support their specification as advocated in [3].

The rest of paper proposes an approach able to support semantic conflicts. In particular, the proposal is based on a meta-model to specify conflicts and corresponding resolution criteria, and to (partly) automate their parallel composition.

3 A metamodel for conflict management

The mechanism proposed in this paper is inspired by the work in [9], where the authors introduce the Join Point Description Diagrams (JPPDs) which are a mechanism to locate modification points in the source code (called *join points* according to aspect-oriented software development [10] terminology) from the model abstraction level. Each join point is defined through a set of UML model element patterns, which can be directly mapped toward OCL expressions. In order to minutely pick up source code points such descriptions must be fine-grained, which makes the resulting pattern definition very powerful. In this respect, OCL is widely accepted as a model query language, and its usability and scalability benefits are discussed in [9].

This paper extends and adapts the mentioned mechanism introduced in order to define a conflict specification formalism which relies on a model-based pattern language endowed with constructs for resolution criteria descriptions. In the following, the details of the conflict specification language will be explained together with its semantics given in terms of OCL expressions.

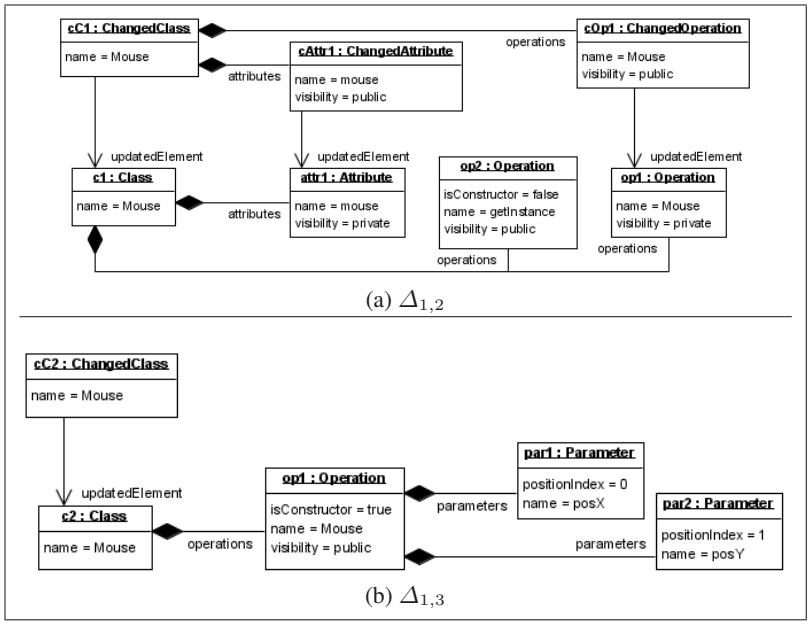

Fig. 5. Sample delta models

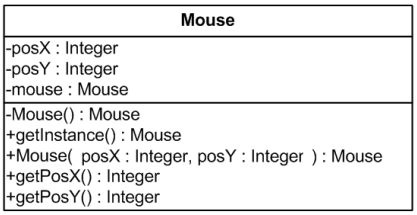

Fig. 6. Sample model $T_{\Delta_{1,3};\Delta_{1,2}}(M_1)$

3.1 Representing Conflicts

Incompatibilities between parallel model modifications are precisely specified by means of *conflict models* relating left- and right- hand sides which represent not allowed contemporary matches. If one of them occurs, a conflicting situation has been found and needs to be solved. The reconciliation can be left to the manual intervention of developers, even if it could become a tedious and error prone task in the context of large systems. Besides, direct manipulation hides the rationale which guided the different choices. Therefore, we made it possible to attach resolution criteria by decorating the links between element patterns with reconciliation decisions.

An excerpt of the conflict specification metamodel is given in Fig. 7: a `Conflict Block` groups a number of pattern boxes and relations between them. In order to have a valid specification at least two patterns and one `ConflictRelation` between them are needed. In other words, conflict models must contain at least a left-hand side and

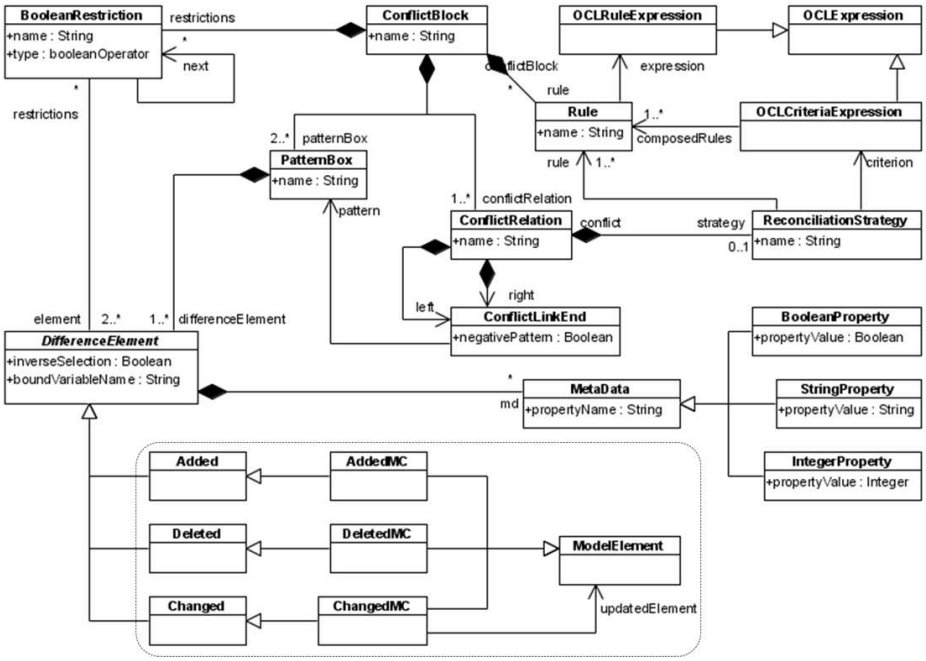

Fig. 7. The conflict specification meta-model.

a right-hand side visualizing an undesired scenario. A `PatternBox` contains a number of `DifferenceElement`s each of which can have some related `MetaData`, i.e. general information which is not intrinsic to the specific difference model like the creation/modification date, the author and so on. Since a difference element represents a pattern, its properties can be used to narrow the matching set if specified; moreover, a variable can be declared which will be bound if there is at least one match (`boundVariableName`). The importance of this variable is that it can be referred to by other patterns through its name; in this way, when a pattern is matched the variable is substituted with the current binding whenever it has been used.

Negative patterns can be specified by means of the `inverseSelection` flag, which can be considered as the `not` boolean operator. Moreover, two or more element patterns can be combined for refinement purposes; by default, they are joined through `AND` relations, i.e. they have all to be matched. Whereas, by means of `BooleanRestriction`s it is possible to compose patterns by using different boolean operators. In this respect, precedences between boolean operators are established through the `next` association. Finally, the `negativePattern` flag in each `ConflictLinkEnd` can be activated to negate the selection of the referred `PatternBox`. When `negativePattern` is enabled (i.e. set to true) an interesting collision management scenario takes place, since it means that there is a problem if a given modification has not been performed. Consequently, mandatory manipulations can also be prescribed through the proposed approach. The conflicting scenarios can be given corresponding resolution criteria through `ReconciliationStrategy`, which states how each conflict should be resolved. Cri-

teria are expressed by means of OCLExpressions [11] and combine Rules, i.e. predicates in terms of Metadata information. In this paper, we focus on conflict specifications only, the interested reader can refer to [12] for a detailed discussion about the definition of reconciliation strategies.

As already mentioned, conflict definitions are based on DifferenceElements. In particular, elements to be related are selected through the corresponding delta meta-classes obtained by means of the automated meta-model generation procedure illustrated in Sect. 2.1. The dashed part in Fig. 7 shows how the current delta language is bound to the pattern. It has to be noted that AddedMC, DeletedMC and ChangedMC entities are used as placeholders of the current derived elements, as for example the AddedClass, ChangedClass and DeletedClass of the simplified UML metamodel shown in Fig. 2. Moreover, three new meta-classes have been added to match all instances of a certain kind of manipulation, that are Added, Deleted and Changed. In summary, delta elements can be selected at different levels of granularity, like the kind of manipulation they represent, the portion of input model entities involved in the modifications and the specific values used to perform the revision.

The definition of the conflict metamodel, as customized for the particular source metamodel taken into account, has been automated through a model transformation (currently implemented in ATL [13]) that takes as input the source metamodel and generates the corresponding conflict specification metamodel. For example, starting from the SimpleUML metamodel in Fig. 2 it is possible to automatically obtain the corresponding SimpleUMLCSpec metamodel, that is the appropriate conflict specification language. The description of the details of such transformation goes beyond the scope of this paper. However, it can be simply obtained by extending the mechanism for the difference metamodel derivation and the interested reader can find it for download at [14].

The obtained SimpleUMLCSpec conflict metamodel can be used to specify conflicts like the one depicted in Fig. 8 through the ConflictBlock element cb. For presentation purposes, the model is given as an object diagram which contains instances of the conflict metamodel even though alternative concrete syntaxes can be defined to have more human readable documents. The model specifies the singleton violation conflict which can be exploited to detect the semantic collision introduced in the previous section. In particular, it presents a collision in which two concurrent updates of a class (cC1 and cC2) bound to the cClass variable, do not converge to compatible results. Going deeper, the left pattern (see the pbLeft element named Singleton) captures the minimal updates required to introduce the singleton design pattern that can be summarized as follows: *(i)* the modifications which involve the visibility of the attribute containing the instances of the class itself, and that of the constructor (see the elements attr1 and opt1, respectively), and *(ii)* the addition of a new public operation which returns instances of the class (see the op2 operation named getInstance).

The right-hand side of the conflict specification (see the pattern box pbRight named Singleton violation) specifies some modifications which corrupt the requirements prescribed for the singleton design pattern. Hence, neither the constructor should be made public (see the cOp2 operation updated with the element op3), nor an additional public constructor should be added (see op4), nor the shared instance value should be made modifiable through a public attribute (see attr2). Such changes are grouped by

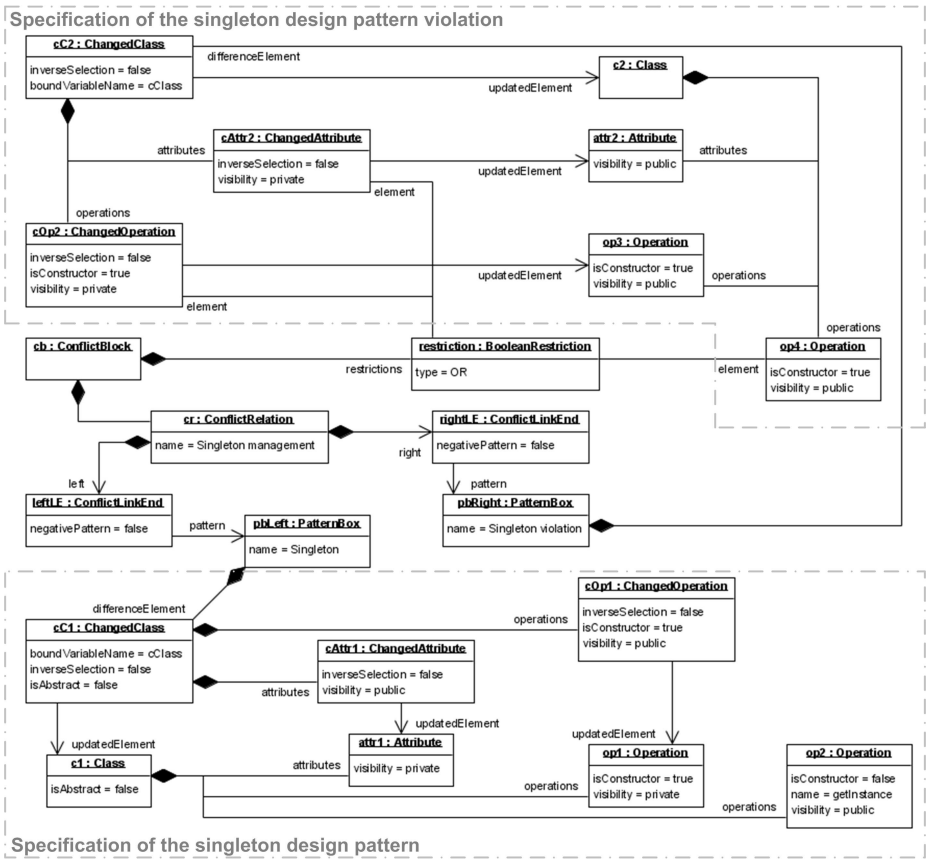

Fig. 8. A model specifying the singleton violation conflict.

means of the `OR` boolean operator represented by the `restriction` element. As said before, pattern entities are glued through an `AND` by default, which would require the contemporary matching of all the conditions. In this case it is sufficient that one of the three different updates is matched to violate the singleton pattern, or in other words to cause a collision with the left-hand side pattern box.

It is worth noting that this conflict definition abstracts from a particular class, whereas it refers to a generic situation where the singleton design pattern is introduced and in the mean time some concurrent delta corrupts that specific pattern. In this respect, it becomes evident the usefulness of bound variables, which can be exploited to refer to the particular values of the current matches (see the property `boundVariableName` of the elements `cC1` and `cC2`).

The application of the model in Fig. 8 to the difference models in Fig. 5 detects the occurrence of a semantic conflict consisting of the singleton violation introduced by the model $\Delta_{1,3}$ with respect to the differences represented in $\Delta_{1,2}$. In fact, by matching $\Delta_{1,2}$ with the left pattern box in Fig. 8 and $\Delta_{1,3}$ with the right one, a semantic conflict

is identified and human interventions are required instead of generating models like the one in Fig. 6 (as current syntactical approaches do) which does not adhere to the singleton prescriptions.

The syntactic description of collisions has to be assisted by its semantics, in the same way difference metamodels induce transformations on input models. Therefore, in the following it is explained how to give precise meanings to conflict declaration models.

3.2 Interpreting Conflict Models

The interpretation of conflict models induces the verification of co-existing updates in the input difference models Δ_1 and Δ_2. Difference element patterns can be seen as model queries which are naturally expressed by OCL expressions, therefore, each model conforming to the metamodel shown in Fig. 7 can be translated into the corresponding OCL constructs (in a similar way to the approach given in [9]), which interrogate the couple of concurrent manipulations to detect collisions and eventually to give them a reconciliation. It has to be noted that the matching procedure has to be founded on a minimal set of expected entities and related properties, which are provided by the meta metamodel.

Let `ml` and `mr` be the instances for Δ_1 and Δ_2 models, respectively. Then the general OCL rule to match patterns against input models can be formulated as follows:

```
1  context ConflictRelation::
2    matchingPatternBoxes(ml: Sequence(DifferenceElement),
3                         mr: Sequence(DifferenceElement)) :
4    Sequence(TupleType(matchedL : Sequence(DifferenceElement),
5                       matchedR : Sequence(DifferenceElement)))
6  post: result =
7    self.matchPBL(ml,self.left)->
8      iterate(
9        lm; res = Sequence {} |
10         let rightMatches = self.matchPBR(lm,mr,self.right) in
11           if (rightMatches->notEmpty())
12           then
13             res->
14               append(Tuple{matchedL = lm, matchedR = rightMatches})
15           else
16             res
17       endif
18     )
```

Listing 1.1. The general OCL rule to match conflicting changes in parallel delta composition.

The function `matchingPatternBoxes` returns a collection of conflicting situation pairs by means of tuples; in turn, each tuple contains a sequence of elements resulting either from the left pattern (`matchedL`) or from the corresponding right pattern selection (`matchedR`). Firstly, left pattern matches are detected through the `matchPBL` function (see line 6); then, this collection of elements is exploited to look for matches on the right-hand side of conflict specification (lines 7-14). In particular, `matchPBR` is iterated on left matchings to detect corresponding elements on the right; if the returned set of elements is not empty, the computed pair of changes is appended as an entry in the current result. In this way, left selections allow to resolve variable bindings which have

been eventually exploited in the right part too, while the right pattern ones are computed autonomously and are not intended to be used on the left-hand side. For example, the `cClass` variable shown in Figure 8 is bound at the beginning of the matching process and then its value is exploited to find the remaining correspondences. In this respect, each time a variable is bound its value is resolved wherever it has been referred to.

Both the function `matchPBL` and `matchPBR` detect correspondences in the same way, by starting with variables which need to be bound (if there exist) and then resolving the remaining portion of correspondences. Listing 1.2 shows a key OCL function called by `matchPBL` and `matchPBR` for selecting entities which match against a given difference pattern.

```
1  context DifferenceElement :: matches(de : Sequence{DifferenceElement}) :
2                               Sequence(DifferenceElement)
3  pre : negate : Boolean = self.inverseSelection and
4        failed : Boolean = false
5  post: result =
6    de->
7      select(
8        mEl |
9          (mEl.oclIsTypeOf(self.oclType()) and
10         (mEl.isAbstract = self.isAbstract or
11          self.isAstract.oclIsUndefined) and
12         (mEl.name = self.name or
13          self.name = isUndefined)) xor
14         negate@pre
15     )->
16       iterate(
17         mEl; result = Sequence {} |
18           self.structuralFeatures->
19             iterate(
20               sF1; res = Sequence {} |
21                 if (sF1.oclIsTypeOf(Reference))
22                 then
23                   mEl.structuralFeatures->
24                     select(
25                       sF2 |
26                         sF2.oclIsTypeOf(Reference) and
27                         (sF1.matchReference(sF2) xor
28                          negate@pre) and
29                         (not sF1.opposite.oclIsUndefined() implies
30                         ((sF1.opposite.matchReference(sF2.opposite) xor
31                          negate@pre))) and
32                         (sF1.type.matches(sF2.type)->notEmpty())
33                     )->
34                       res->collect({sF2, sF1.type.matches(sF2.type)})
35                   and
36                     res->isEmpty() implies failed = true
37                 else
38                   res->collect(sF1.matches(de.structuralFeatures)) and
39                   res->isEmpty() implies failed = true
40             ) and
41               if (not failed)
42               then result->collect({mEl,res})
43               endif
44       )
```

Listing 1.2. The general OCL rule to match a delta element pattern.

More precisely, given a difference element pertaining to the model specifying conflicts, the `matches` OCL function looks for possible correspondences in the input difference document provided as parameter. Therefore, firstly potential matching meta-classes are selected (lines 7-14) and then the related set of structural features needs to be met (lines

16-43). Going into more details, for each outgoing `Reference` both its properties and the linked element have to match (lines 21-35). In an analogous manner, the remaining structural properties (i.e. attributes) have to be considered as new patterns to be matched through recursive calls (lines 37-38).

At the end of the outlined process, a sequence is returned; if it is empty then the matching computation failed, i.e. no conflicts have been found, otherwise it contains entities in the difference document which correspond to the patterns depicted in the conflict model. In this respect, the binding of variables is preserved implicitly by means of the `DifferenceElement` meta-class which allows to enrich a delta element with the declared bound variable.

4 Related works

This paper deals with colliding revisions of artefacts developed in parallel. Especially, it copes with interferences which can not be detected by means of syntactic analyses and are usually referred to as semantic conflicts [3]. Those issues are considered in few works, which tend to provide support only to a particular modeling language (see [15] for a detailed discussion). In this respect, the proposed technique can be considered as an extension of [16] to arbitrary metamodels.

In [15] the authors illustrate *SMoVer*, that is a model versioning tool able to deal with both syntax (structural) and semantic (behavioural) conflicts. In particular, the semantic issues are faced by exploiting appropriate view definitions toward which structural specifications are mapped. In other words, they use translational semantics in order to elicit particular interferences. With respect to our work, the mechanism in [15] requires new domain characterizations to customize the set of the identified interferences. Moreover, conflicts are detected by comparing the whole concurrent revisions, whereas here collisions are managed taking into account only difference documents, i.e. only the elements involved by changes.

Composition of differences is partly considered in few works which mainly focus on software refactoring like [8] and differences occurring on systems developed in parallel [17]. In [8] the authors propose a graph-based technique to discover couples of manipulation conflicts (*critical pairs*). Then, a set of conflict reduction heuristics is given, even if developers can not specify their own resolution criteria. The work in [17] deals with differences occurring on systems developed in parallel; through the use of program slicing and system dependency graphs, semantic interferences are detected. However, the level of abstraction is too low with respect to models. In this respect, we presented an approach to raise the level of abstraction by reformulating part of the knowledges and experiences of these works in a model-driven setting.

A number of efforts deal with the general problems of consistency management and model composition; as software systems are specified by means of disparate notations related to different points of view of the same application, keeping the different documents coherent with each others is an unavoidable issue to take care of [18, 19]. In particular, consistency management is faced by defining a kind of conflicts between modifications occurring on the different views which are solved by propagating the changes toward newer coherent documents. However, in general these solutions are

specifically designed for the problem domain they are applied in, like the UML diagrams and similar. Moreover, inconsistencies are always thought as existing between a pair of elements, while conflicts can be defined as ranging from a single and independent entity update to an arbitrary group of element manipulations. In summary, consistency management can be considered as a sub-problem of conflict issues.

In [20] a set of common definitions are provided to outline the key requirements of a model composition solution, in terms of language and tool support. Moreover, in [21] a number of operators for model integration are described and they have partially inspired the composition constructs proposed so far. Finally, other works take into account the specific issue of model merging, both in the case of different versions of a system design [22] and in a generic scenario, like [23, 24] for example. However, those papers seem to focus on establishing correspondences between elements of the input models (matching phase), while conflict troubles are largely missed out.

5 Conclusions and future work

Increasingly, complex software systems are cooperatively designed in distributed environments and the interaction among concurrent manipulations inevitably causes conflicts which must be detected and reconciled.

This paper illustrated a model-driven conflict specification mechanism based on the declarative description of incompatibilities between competing difference models. In particular, a conflict metamodel has been proposed to specify conflict patterns between difference elements which cannot take place at the same time. This way, it is possible to introduce non-syntactical conflicts which usually would not be identified by merging approaches based on implicit conflict detection techniques. As a benefit, it is possible to improve conflict management adaptability to different application domains, since each domain has a set of its own semantic dependencies which need to be specified to obtain a useful detection and resolution method.

Future work will necessarily encompass further validation against large projects in order to evaluate the efforts demanded to the developer for managing model compositions. For example, the process could benefit of improved automation degree by taking into account heuristics and by-example approaches [15]. Finally, an enhancement of the conflict specification language will be investigated for supporting multi-view consistency, which demands the description of change interferences between models not necessarily conforming to the same metamodel, and hence inducing disparate difference metamodels.

References

1. Whitehead, J.: Collaboration in software engineering: A roadmap. In: FOSE '07: 2007 Future of Software Engineering, Washington, DC, USA, IEEE Computer Society (2007) 214–225
2. Favre, J.M.: Meta-Model and Model Co-evolution within the 3D Software Space. In: Procs. of the Int. Workshop ELISA at ICSM. (September 2003)
3. Mens, T.: A State-of-the-Art Survey on Software Merging. IEEE Trans. Softw. Eng. **28**(5) (2002) 449–462

4. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A Metamodel Independent Approach to Difference Representation. Journal of Object Technology **6**(9) (October 2007) 165–185
5. Lin, Y., Zhang, J., Gray, J.: Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In: OOPSLA Work. MDSD. (2004)
6. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Model Differences for Supporting Model Co-evolution. In: Procs. MoDSE, 2nd Workshop on Model-Driven Software Evolution. (2008) To appear.
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley (1995)
8. Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. Electr. Notes Theor. Comput. Sci **127**(3) (2005) 113–128
9. Stein, D., Hanenberg, S., Unland, R.: A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models. In: MDAFA'03 and MDAFA'04. Volume 3599 of LNCS., Springer-Verlag (2005) 77–92
10. Filman, R., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley (2004)
11. Object Management Group (OMG): OCL 2.0 Specification (2006) OMG Document formal/2006-05-01.
12. Cicchetti, A.: Difference Representation and Conflict Management in Model-Driven Engineering. PhD thesis, University of L'Aquila, Computer Science Dept. (2008)
13. Jouault, F., Kurtev, I.: Transforming Models with ATL. In Bruel, J.M., ed.: MoDELS Satellite Events. Volume 3844 of LNCS., Springer-Verlag (2005) 128–138
14. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Management of conflicts on the AMMA platform. http://www.di.univaq.it/cicchetti/conflictManagement.php (May 2008)
15. Altmanninger, K., Bergmayr, A., Schwinger, W., Kotsis, G.: Semantically Enhanced Conflict Detection between Model Versions in SMoVer by Example. In: Procs of the Int. Workshop on Semantic-Based Software Development at OOPSLA 2007, Montral, Canada. (October 2007) to appear.
16. Cicchetti, A., Rossini, A.: Weaving models in conflict detection specifications. In: Procs of the ACM Symposium on Applied Computing (SAC '07), Model Transformation track, New York, NY, USA, ACM (2007) 1035–1036
17. Thione, G.L., Perry, D.E.: Parallel Changes: Detecting Semantic Interferences. In: COMPSAC, IEEE Computer Society (2005) 47–56
18. Egyed, A.: Fixing Inconsistencies in UML Design Models. In: Procs. of the 29th ICSE 2007, IEEE Computer Society (2007) 292–301
19. Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency Management with Repair Actions. In: Procs. of the 25th ICSE 2003, May 3-10, IEEE Computer Society (2003) 455–464
20. Bézivin, J., Bouzitouna, S., Fabro, M.D.D., Gervais, M.P., Jouault, F., Kolovos, D., Kurtev, I., Paige, R.: A Canonical Scheme for Model Composition. In Rensink, A., Warmer, J., eds.: Procs of the 2nd ECMDA-FA. Volume 4066 of LNCS., Springer (2006) 346–360
21. Reiter, T., Kapsammer, E., Retschitzegger, W., Wimmer, M.: Model Integration Through Mega Operations. In: Workshop on MDWE 2005. (Jul 2005)
22. Kolovos, D.S., Paige, R.F., Polack, F.: Merging Models with the Epsilon Merging Language (EML). In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Procs. of MoDELS 2006, Genova, Italy, October 1-6. Volume 4199 of LNCS., Springer (2006) 215–229
23. Engel, K.D., Paige, R.F., Kolovos, D.S.: Using a Model Merging Language for Reconciling Model Versions. In Rensink, A., Warmer, J., eds.: Procs of the 2nd ECMDA-FA. Volume 4066 of LNCS., Springer (2006) 143–157
24. Pottinger, R., Bernstein, P.A.: Merging Models Based on Given Correspondences. In: VLDB. (2003) 826–873