JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2007

Vol. 6, No. 9, 2007

A Metamodel Independent Approach to Difference Representation

Antonio Cicchetti, **Davide Di Ruscio**, and **Alfonso Pierantonio**
Università degli Studi dell'Aquila, Italy

It is of critical relevance that designers are able to comprehend the various kinds of design-level modifications that a system undergoes throughout its entire life-cycle. In this respect, an interesting and useful operation between subsequent system versions is the model difference calculation and representation.
In this paper, a metamodel independent approach to the representation of model differences which is agnostic of the calculation method is presented. Given two models which conform to a metamodel, their difference is conforming to another metamodel derived from the former by an automated transformation. Difference models are first-class entities which induce transformations able to apply the modifications they specify. Finally, difference models can be composed sequentially and in parallel giving place to more complex modifications.

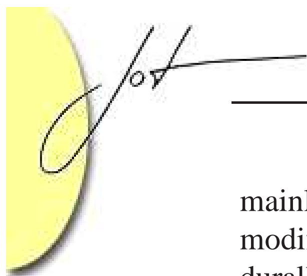
1 INTRODUCTION

Model-Driven Engineering (MDE) leverages models to first-class status by shifting the focus of software development from coding to modeling. It is of critical relevance that designers are able to comprehend the various kinds of design–level modifications that a system undergoes throughout its entire life-cycle. Nurturing the detection of differences between models is essential to model development and management practices, which are traditionally not neglected in high-quality software development processes [10].

There have been some work (e.g., [22, 1, 29]) that proposed automated UML–aware differencing algorithms which, in contrast with traditional lexical approaches, such as *GNU diff-like* tools (see [12, 13, 14] among others), are capable of capturing the high-level logical/structural changes of a software system. More recently, another approach [20] based on structural similarity extended differencing to metamodel independency, i.e., to models conformant to an arbitrary metamodel. However, the capability of tools to operate on change documentation which conforms only to their own internal format tends to lock software development into a single tool compromising its exploitation as part of a tool chain. In fact, whilst a number of algorithms and tools are available for detecting structural changes, the visualization of differences is often based on solutions where the opportunity to harness the power of generic modeling platforms has far been largely missed.

At the moment, the techniques for visualizing and representing model differences are

Cite this column as follows: Antonio Cicchetti, Davide Di Ruscio, Alfonso Pierantonio: *A Metamodel Independent Approach to Difference Representation*, in Journal of Object Technology, vol. 6, no. 9, 2007, pages 165–185,
http://www.jot.fm/issues/issues_2007_10/paper9



mainly based on edit scripts [1, 21] and coloring techniques [22]. The former represents modifications as a sequence of atomic actions specifying how the initial model is procedurally modified. Whereas, the latter permits the differences to be displayed in a diagram which is the union of the two base models, with the common parts of both base diagrams painted black and the specific elements colored. Unfortunately, to different extent both solutions present drawbacks not limited to a certain lack of abstraction and compositionality which compromises their adoption in a generic modeling platform [3]. In fact, edit scripts are intrinsically not declarative, lengthy and very fine-grained, suitable for inner representations but quite ineffective to be adopted for documenting changes (e.g., see [7]). Coloring techniques presents advantages over procedural methods, for instance differences are given as a model which enhances intuitiveness and can provide the basis for a variety of subsequent analysis. However, they tend to be densely populated, require dedicated tool support, and subsequent difference calculations are not compositional.

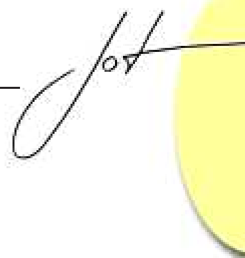
In this paper, we present a metamodel independent approach to the representation of model differences which is agnostic of the calculation algorithm, i.e., the proposed techniques do not refer to any differencing methods nor tools and aim at providing a mean to represent version changes. Given two models which conform to an arbitrary metamodel, their difference conforms to another metamodel derived from the former by an automated transformation. Interestingly, difference models are first-class objects which induce transformations, such that they can be applied to one of the differenced models to automatically obtain the other one. This operation can be, under certain conditions, composed sequentially and/or in parallel in order to represent more complex modifications.

The paper is structured as follows: Sect. 2 describes the current approaches to model difference representation and visualization and outlines a minimal set of requirements a representation technique should, in our opinion, satisfy. Next section presents the proposed approach by defining an automated transformation from an arbitrary metamodel to a corresponding difference metamodel. How a difference model induces, in turn, a transformation is given in Sect. 4 by means of a higher-order transformation. Sect. 5 introduces the dual, parallel and sequential composition operators for differences. Finally, after discussing some related work we draw some conclusions.

2 BACKGROUND

The rationale behind the design–level modifications that a system undergoes during its life-cycle is of key relevance in model development and management practices. Detecting differences and identifying mappings among distinct versions of a system design is preparatory to represent at least part of such knowledge. The more the documents increase in intricacy, the more specialized tools are needed to compare, manage, and represent the different models into a new one that contains all the proposed changes.

As mentioned, we are interested in finding a suitable representation for model differences which abstracts from the calculation method and permits to harness the potential offered by generic modeling platforms (for instance [4, 19]). Thus, we identified a number of natural properties a representation technique should have according to our view, as



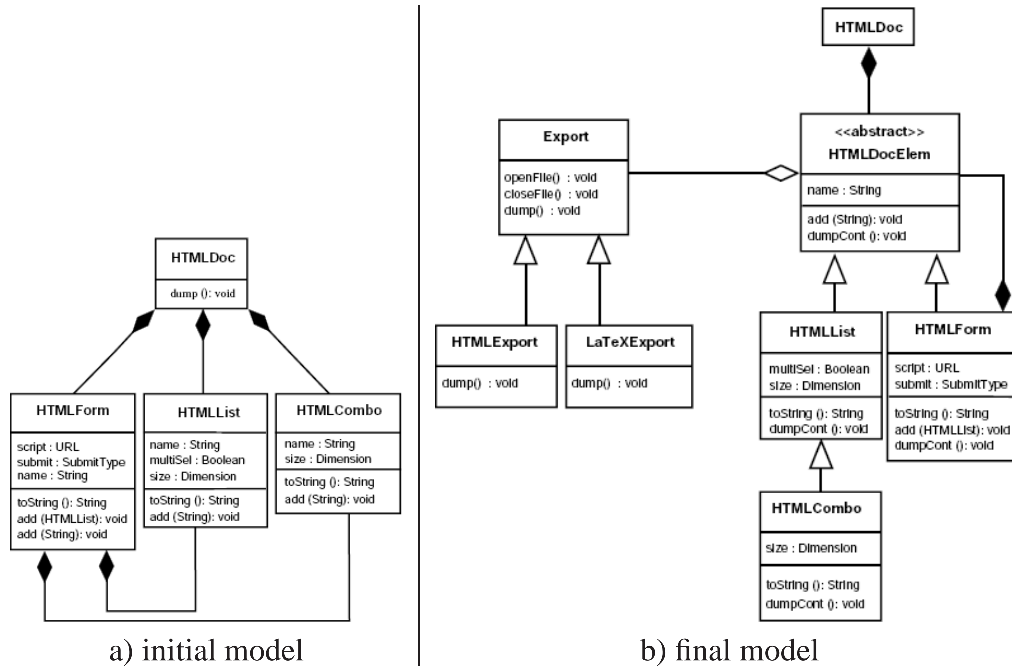
described below

– *model-based*, the outcome of a difference calculation must be represented as a model to conform to the spirit of “everything is a model” principle [3] and to enable a wide range of possibilities, such as subsequent analysis, conflict detection or manipulations;
– *minimalistic*, the difference model must contain only the necessary information to represent the modifications, without duplicating parts as those model elements which are not involved in the change;
– *transformative*, each difference model must induce a transformation, such that whenever applied to the initial model yields the final one. Moreover, the transformation must be applicable also to any other model which is possibly left unchanged in case the elements specified in the difference model are not contained in it;
– *compositional*, the result of subsequent or parallel modifications is a difference model whose definition depends only on difference models being composed and is compatible with the induced transformations;
– *metamodel independent*, the representation techniques must be agnostic of the *base* metamodel, i.e., the metamodel the base models conform to. In other words, it must be not limited to specific metamodels, as for instance happens for certain calculation methods (e.g., [22, 29]) which are given for the UML metamodel.

The above discussion outlines a minimal set of requirements which should be taken into account in order to let a generic modeling platform deal with advanced model management facilities. In the rest of the section, the most common visualization techniques are compared according to a small benchmark case borrowed from [22] and illustrated in Fig. 1. In particular, in the final model export functionalities have been added to the initial model through the `Export` class; consequently an abstract `HTMLDocElem` has been created, which is specialized by `HTMLList` and `HTMLForm`. In turn, `HTMLList` is specialized by `HTMLCombo`, while `HTMLForm` is composed by `HTMLDocElems`, respectively. These modifications are intended to be manually performed on the initial model and can be detected by means of an automated tool implementing one of the existing differencing algorithms. The visualization of differences can be divided into two main techniques: *directed deltas* and *symmetric deltas* [21]. The former represents delta documents as the sequence of the operations needed to obtain the new version from the old one, while the latter shows the results as the set difference between the two compared versions. In the sequel, a deeper description of both techniques will be provided by means of the example introduced above, aiming at highlight pros and cons about each of them.

Edit Scripts

Edit scripts represent an implementation of the directed delta approach. Sequences of primitive operations, like `add`, `edit` and `delete` for instance, describe in procedural terms the modifications a model has been subject to. In general, such technique is strictly



a) initial model b) final model

Figure 1: Different versions of a system design

related to the calculation algorithm because of optimization issues, such as unexpected redundancy [21]. In fact, if on one hand the calculation is based on a set of atomic operations which is independent from any differencing method, on the other hand the optimization requires the calculation to provide a precise ordering of the operations.

A major advantage of this techniques is the compositionality, i.e., the capability of obtaining a document, which underwent a number of subsequent modifications, by applying the composition of deltas to the initial document. This quality factor combined with the optimization makes the technique very appreciated for its efficiency. Unfortunately, the readability and intuitiveness of the outcome result is limited, especially when the scripts are largely optimized and the rationale behind the updates tends therefore to be blurred. Additionally, the calculation method proposed in combination with edit scripts typically identifies elements among distinct versions of a model by means of persistent identifiers. Consequently, delta documents result locked within the tool which has been used for entering/editing the base models and the opportunity of having *transformative* deltas is largely missed.

Coloring techniques

Coloring techniques permit the modifications to be displayed in a diagram which is the union of the two base models, with the common parts of both base diagrams painted black and the specific elements denoted by colors, tags, or symbols, respectively. It is a symmetric delta approach, since it directly compares two versions of a model and highlights the changes which took place. In [22] a calculation and visualization method based on

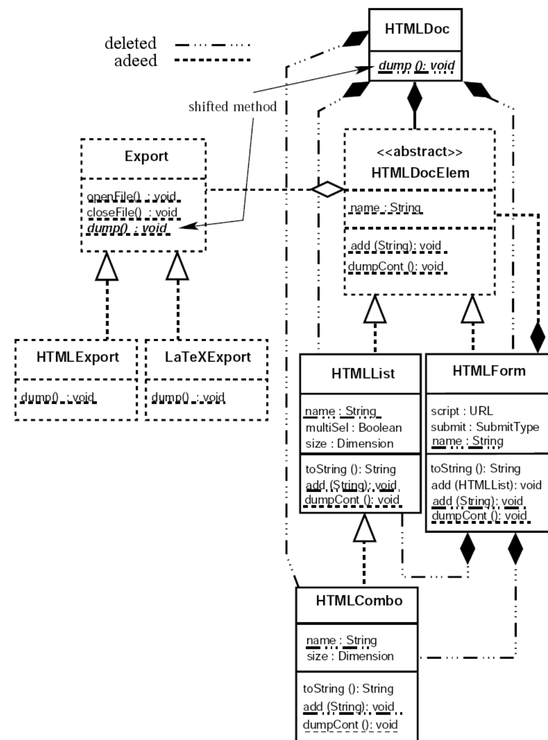


Figure 2: An example of difference visualization technique.

coloring is proposed and Fig. 2 depicts the resulting delta document applied to the base models in Fig. 1. Alternative representations are possible and usually based on element stereotyping and delta tree arrangements [29] among others. Visualizing modifications according to this technique is typically beneficial for the designer, since the underlying rationale can be grasped with a glance thanks to enhanced intuitiveness and readability. However, these quality factors are retained only if the base models are not large and not too many updates apply to the same elements, since the difference model consists of both base models to denote the differences. Finally, this causes the method not to be *transformative* property as well.

3 DIFFERENCE METAMODEL

In this section, we propose an approach to model differences capable to meet the requirements discussed in Sect. 2. For presentation purposes, the simplified UML metamodel in Fig. 4 is considered throughout the section although the approach is general and is applicable to any metamodel. According to the “everything is a model” principle [3], this work proposes an approach to specify differences as models and that can be taken as input by general purpose theories and tools in a MDE setting. In particular, in MDE models are not considered as merely documentation but precise artifacts that can be understood by computers and can be automatically manipulated. In this scenario, metamodeling plays a key role: it is intended as a common technique for defining the abstract syntax of models

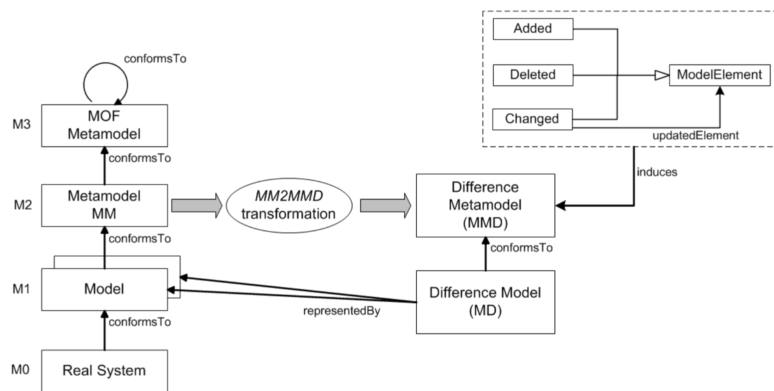


Figure 3: Difference metamodel generation

and the interrelationships between model elements. Metamodeling can be seen as the construction of a collection of *concepts* within a certain domain formalized in a metamodel which describes the common properties of its instances, i.e., models which represent abstractions of real world phenomena. A model is said to *conform to* its metamodel like a program conforms to the grammar of the programming language in which it is written [3]. In this respect, the four-level architecture illustrated on the left hand side of Fig. 3 describes the *conformance* relation: at the bottom level, the M0 layer is the real system. A model represents this system at level M1; this model conforms to its metamodel defined at level M2 that in turn conforms to the meta–metamodel at level M3. The meta-metamodel conforms to itself.

Provided that, given two models being differenced and that conform to a given metamodel *MM*, their difference conforms to another metamodel *MMD* that can be automatically derived from *MM*. In particular, the new metamodel has to provide the constructs able to express the modifications that have to be performed on the initial version of a given model in order to obtain the final one. The proposed approach permits the representation of changes that can classified as follows:

– *additions*: new elements are added in the final model like the `HTMLDocElem` abstract class in Fig. 1.b not present in the initial version of the specification;

– *deletions*: some of the existing elements are deleted as a whole like the composition relation between the `HTMLDoc` and `HTMLList` classes in Fig. 1.a;

– *changes*: a new version of the model can consist of some updates of already existing elements. For example, some structural features (attributes and operations) of the `HTMLList` class in Fig. 1.a have been modified giving place to the new version in Fig. 1.b.

In particular, let MC be a metaclass of a given metamodel, then it defines the AddedMC, DeletedMC and ChangedMC metaclasses that enable the representation of additions, deletions and changes, respectively, of elements conforming to the MC metaclass (see the right hand side of Fig. 3). For instance, the metaclasses `AddedClass`, `DeletedClass` and `ChangedClass` in Fig. 5 are derived by the metaclass `Class` depicted in Fig. 4. The

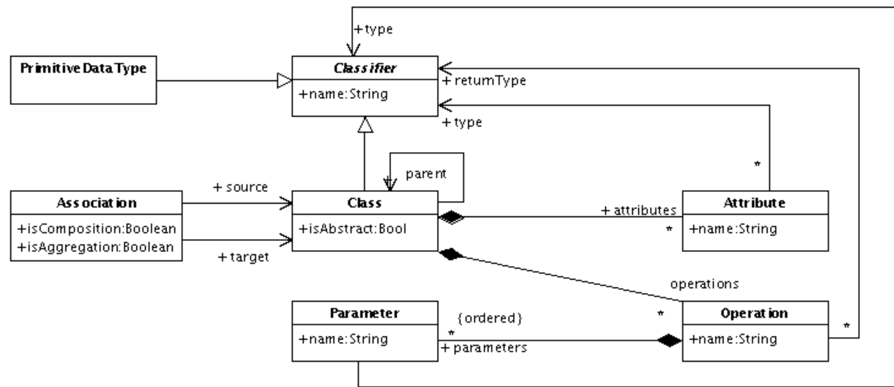


Figure 4: Sample UML metamodel

resulting metamodel allows the representation of the differences among two distinct versions of a UML model, as in Fig. 7 where only part of the differences between the two versions in Fig. 1 are reported. For example, the `HTMLDocElem` class in Fig. 1.b is represented as an `AddedClass` instance since it is not present in the initial version of the model in Fig. 1.a. The deletion of elements is represented by means of instances of the corresponding Deleted metaclass like the composition association between the `HTMLDoc` and `HTMLList` classes which is represented as an instance of the `DeletedAssociation` metaclass. Whenever the deletion concerns a container, the metamodel prescribes that also its contained elements must be denoted as deleted, although this may appear redundant if not counterintuitive. The motivation is that a difference model must be a self–contained unit, i.e., in case the internal elements of a deleted container are not explicitly marked as deleted, such information could only be deducted by navigating the initial model. This is shown to be relevant in Sect. 5 when discussing the dual notion.

Changes of already existing elements are represented through Changed elements as the class updates which are given by means of `ChangedClass` instances each of them associated with a corresponding `updatedElement` class. The latter specifies how `ChangedClass` has to be modified in the new model version in terms of attributes and associations. The Changed modification is kind of shortcut which groups simple modifications consisting of Added and Deleted only reducing the size of the overall difference model. For example, the modified `HTMLList` class in Fig. 7 is composed by the attribute `name` and the operation `add`; both features are not present in the `updatedElement` class which consists of the `dumpCont` operation only. This means that all the structural features which are given in the `ChangedClass` instance but not in the associated `updatedElement` will be deleted in the new version. The features which are not represented in the `ChangedClass` instance will remain unchanged and will be simply copied in the new version of the given element (like the operation `toString` or the attribute `multiSel` in the `HTMList` class). Finally, the features specified both in the `ChangedClass` instance and in the associated `updatedElement` will be modified according to descriptions given in the last one.

Modifications occurring in ordered references (e.g., `Parameter` metaclass in Fig. 4) require to be treated with some additional support. More in detail, the *ordered* association

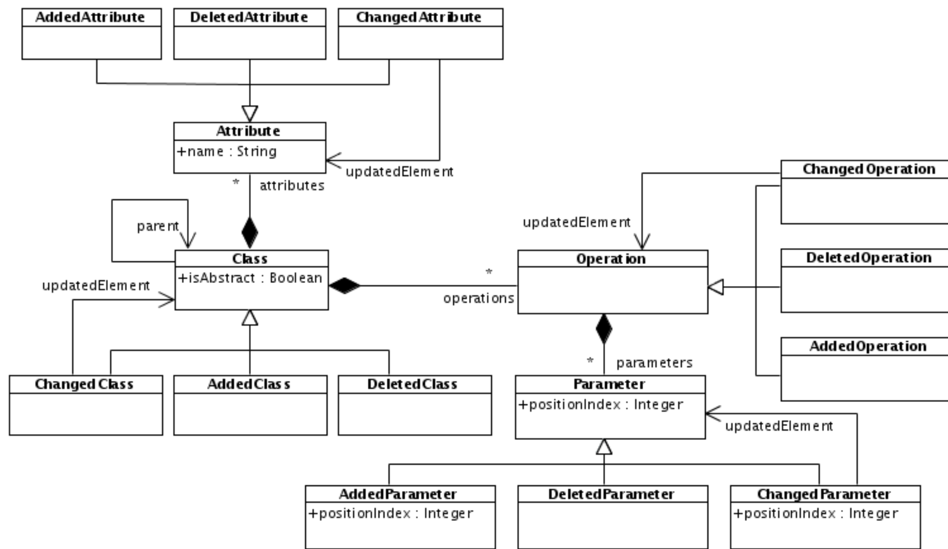


Figure 5: Fragment of the extended UML metamodel

ends induce in all the metaclasses of the difference metamodel but the `Deleted` ones an additional attribute called `positionIndex`. This enables the management of ordered sets in terms of absolute positioning. For instance, let us suppose to have an initial UML class having the operation `op(a:int,b:int)` and due to a manual intervention, the parameter order changed giving place to `op(b:int,a:int)`, then the corresponding difference model (restricted to the only operation `op`) is in Fig. 6.

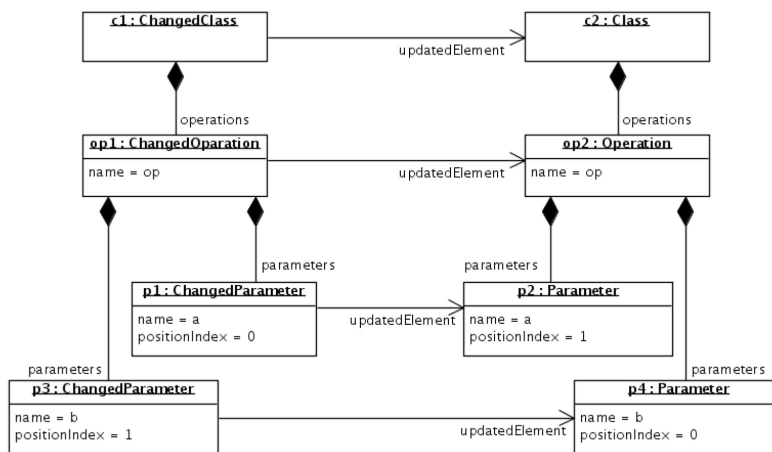
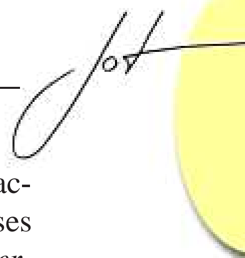


Figure 6: Sample ordering difference

As a side remark, please consider that analogously to the previous container situation also in this case the class the operation is defined in must be denoted as "changed" as well.

As previously pointed out, the approach is metamodel independent and a given metamodel can be automatically extended with the metaclasses needed to represent the modifications among base models that conform to that metamodel. The discussion given



above about the Added, Deleted and Changed concepts has been done by taking into account the Class element of the sample UML metamodel in Fig. 4 for descriptive purposes only. However, the explained behavior is valid for any metamodel and the resulting *genericity* permits the specification of a canonical metamodel extension. In fact, according to the general picture in Fig. 3, a model transformation *MM2MMD* to yield the difference metamodel *MMD* associated with *MM* must be defined. Introducing new metaclasses to denote updated, deleted, and added model elements it is not the only way of modeling modifications, but provides a mean for mapping them to any concrete syntax, as tagging or annotations. Besides, specialization conceptually groups the kind of modifications related to a base metaclass and makes the transformations (which are applied to the difference model) simpler in their formulation.

In the current implementation (available for download at [9]), the *MM2MMD* transformation is given in ATL [18], a QVT compliant language part of the AMMA framework [4]. Due to space limitation, Listing 1 illustrates only a fragment of the implementation. In particular, we reported those rules which modify the source metamodel w.r.t. the structure shown in Fig. 3 disregarding the simple functionalities, such as copying from the source to the target metamodel.

ATL is a hybrid language which contains a mixture of declarative and imperative constructs. Transformation definitions in ATL consist of `modules` each containing a `header` section, `import` section, and a number of *helpers* (that will be described in the next section) and *transformation rules*. The header contains declarations, such as the module name, the source and target models (lines 1-2) with their typing metamodels. The keyword `create` denotes the target model, whereas the keyword `from` indicates the source one. In the following code, the source and target metamodels are both KM3 [16] which is a metamodeling language part of the AMMA framework and based on the same core concepts used in OMG/MOF [23] and EMF/Ecore [5]: classes, attributes and references. In other words, the current implementation is an endogenous transformation over KM3 metamodels.

```
1  module Metamodel2MetamodelDiff;
2  create OUT : KM3 from IN : KM3;
3  ...
4  rule Class2ClassDiff {
5    from
6      s : KM3!Class
7      (
8        not s.isAbstract
9      )
10
11   to
12     t : KM3!Class ( --topClass
13       name <- s.name,
14       ...
15     ),
16
17     a : KM3!Class ( --addedClass
18       name <- 'Added'+s.name,
19       ...
20     ),
21
22     d : KM3!Class ( --deletedClass
23       name <- 'Deleted'+s.name,
```

```
24        ...
25
26      ),
27
28      c : KM3!Class ( --changedClass
29        name <- 'Changed'+s.name,
30        ...
31      ),
32
33      ass : KM3!Reference ( --updatedElement reference
34        name<-'updatedElement',
35        owner<-c,
36        ...
37        type <-t
38        )
39  }
```

Listing 1: Fragment of the *MM2MMD* transformation

Helpers and named rules are the constructs used to specify the transformation functionality; relations between *source* and *target patterns* are given as declarative rules, called *matched rules*. In particular, the source pattern of the rule (lines 5-9) consists of a *source type* and a OCL [24] *guard* stating that only non abstract classes must be matched. The target pattern (lines 11-39) is composed of a set of *elements*, each of them (as the one at lines 12-15) specifies a *target type* from the target metamodel (for instance, the type `Class` from the `KM3` metamodel) and a set of *bindings*.

A binding refers to a feature of the type, i.e., an attribute, a reference or an association-end, and specifies an expression whose value initializes the feature. The elements `a`, `d`, and `c` of the target pattern (lines 11-39) are devoted to the generation of the added, deleted and changed sub-classes of the matched source class `s`, respectively. Finally, the reference `ass` is created as a structural feature of the element `c` in order to provide with the possibility to refer to the new version of a given changed class by means of the `updatedElement` reference. Difference models are first-class artifacts which, in turn, induce other transformations, such that they can be applied to one of the differenced mod-

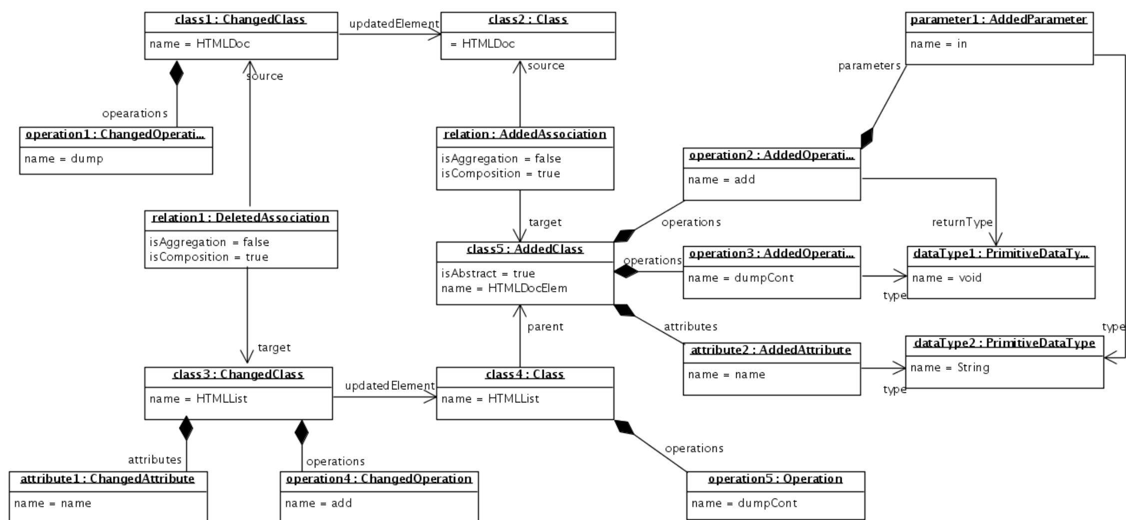


Figure 7: A difference model fragment

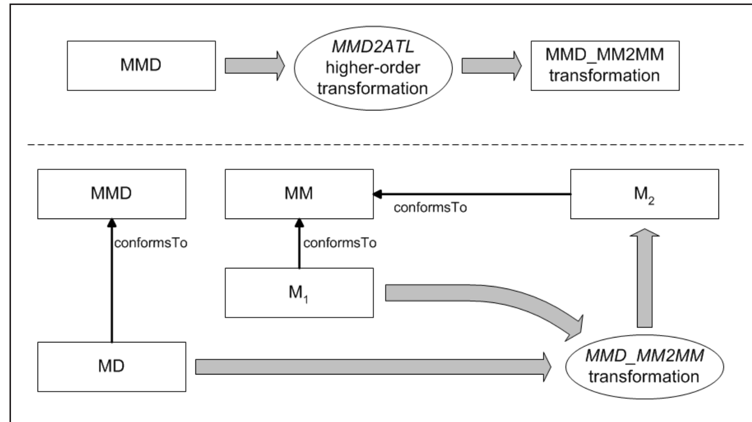


Figure 8: Difference application

els to automatically obtain the other one. Next section describes the techniques behind such a *transformative* quality of the illustrated difference models.

4 DIFFERENCE APPLICATION

The transformative property introduced in Sect. 2 denotes the important capability to employ modifications by interpreting the difference model specifying them. The difference application is twofold: it can be used to "reconstruct" the final model starting from the initial one, but it can also be applied to any model conforming to the base metamodel giving possibly place to an idempotent application in case it does not overlap the initial model. In summation, difference models can be viewed as *patches* operating over models, even though the induced transformations are somewhat exact, i.e., do not comprises any fuzziness factor or adjustability of their application (see Sect. 7 for a discussion about that).

The model difference interpretation is intrinsically difficult since it requires a higher-order transformation, i.e., transformations taking other transformations as input and/or transformations producing other transformations as output [3]. In particular, according to the lower side of Fig. 8, the model transformation (*MMD_MM2MM*) can be applied to a source model M_1 in order to obtain a target M_2 with respect to the differences specified in a model *MD*. Such a model conforms to the metamodel *MMD* automatically obtained from *MM* as discussed in the previous section. More in detail, the *MMD_MM2MM* transformation implements the rules to apply on a model M_1 the *additions*, *deletions* and *changes* specified in the model *MD*. More precisely, considering the dashed part in Fig. 3, for each metaclass MC in the metamodel *MM*, the transformation *MMD_MM2MM* contains the following rules:

– AddedMC2MC: it manages the elements in the difference model *MD* that conform to the AddedMC metaclass. For each element, the rule creates in M_2 a new instance of MC setting the corresponding structural features according to the specification of the AddedMC element;

– ChangedMC2MC: it updates already existing elements in the initial model of type

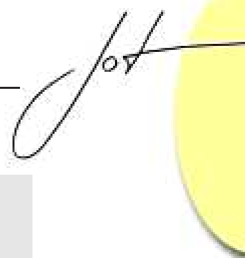
MC according to the modifications specified in *MD* through ChangedMC instances;

– UnchangedMC2MC: it copies the unmodified instances of the metaclass MC which have to be the same both in M_1 and M_2. The source pattern of this rule has a guard matching only the MC elements which have not been changed nor deleted.

Concerning the management of DeletedMC instances, no rules are provided, since the guard in the source pattern of the UnchangedMC2MC rule guarantees that elements which have been specified as deleted in the difference model are not matched during the transformation phase (hence, not copied in the target model M_2).

The following ATL code is a fragment of the *UMLD_UML2UML* transformation that applies on a given UML model the modifications expressed in a difference model (that conforms to the corresponding *UMLD* metamodel like the one in Fig. 5) generating the final UML specification. Due to space limitation, only the code for managing the metaclass `Class` is considered providing the `AddedClass2Class`, `ChangedClass2Class`, and `UnchangedClass2Class` rules that reify the general behaviors of the transformation *MMD_MM2MM* previously illustrated. The transformation rules can use ATL helpers, i.e., read-only functions, to navigate the difference model to find the values to be assigned to the structural features of the new version of a given element. For instance, the new value of the attribute `name` of a changed class (line 28) is reached in the difference model by navigating the `updatedElement` association of the considered changed class. The dedicated helper `getChangedClassname` is used for this purpose and given a changed class it returns the new value for the attribute name (see lines 4-5).

```
1  module UMLD_UML2UML;
2  create OUT : SimpleUML from IN1 : SimpleUML, IN2 : SimpleUMLDiff;
3
4  helper context SimpleUMLDiff!ChangedClass def: getChangedClassname : String =
5    if not self.updatedElement.oclIsUndefined() then self.updatedElement.name else
         OclUndefined endif;
6  ...
7
8  rule AddedClass2Class {
9    from
10     s : SimpleUMLDiff!AddedClass
11     (
12       s.oclIsTypeOf(SimpleUMLDiff!AddedClass)
13     )
14   to
15     t : SimpleUML!Class(
16       ...
17     )
18 }
19
20 rule ChangedClass2Class {
21   from
22     s : SimpleUMLDiff!ChangedClass
23     (
24      s.oclIsTypeOf(SimpleUMLDiff!ChangedClass) and not s.updatedElement.oclIsUndefined()
25     )
26   to
27     t : SimpleUML!Class(
28       name <- s.getChangedClassname,
29       ...
30     )
31 }
```



```
32
33 rule UnchangedClass2Class {
34   from
35     s : SimpleUML!Class
36     (
37       s.oclIsTypeOf(SimpleUML!Class) and not s.isChanged and not s.isDeleted
38     )
39   to
40     t : SimpleUML!Class(
41       ...
42     )
43 }
44 ...
```

Listing 2: Fragment of the *UMLD_UML2UML* transformation

The *UMLD_UML2UML* transformation above has been automatically generated by means of a higher-order transformation applied to the UML difference metamodel shown in Fig. 5 according to the upper side of Fig. 8. Such a generation is feasible since the behaviors of the building blocks of a model difference (that is additions, deletions and changes) are "parametrically" defined and can be instantiated on the elements of a given metamodel. For example, the generic transformation rule ChangedMC2MC described at the beginning of this section can be instantiated on the metaclass `Class` of the UML metamodel producing the `ChangedClass2Class` rule of the *UMLD_UML2UML* transformation. Alternative to a higher-order transformation for generating ATL transformations is serializing models into code by means of a templating mechanism. However, ATL is part a set of coordinated languages and tools, as for instance Textual Concrete Syntax [17] (TCS) devoted to bridging abstract and concrete syntaxes, which make templating for ATL model/code generation unnecessary.

An ATL implementation of this higher-order transformation is available for download at [9], since because of space limitation only a very small fragment can be accommodated in the current work (see Listing 3). The implementation consists of three main rules that are `AddedClass` (lines 4-23), `UnchangedClass` (lines 25-39), and `ChangedClass` (lines 41-52). They are dedicated to the generation of the three kinds of rules needed for the management of each metaclass specified in the source difference model. For instance, the match of the following `AddedClass` rule with the metaclass `AddedAssociation` of the UML difference metamodel, generates in the transformation *UMLD_UML2UML* the rule `AddedAssociation2Association`.

```
1 module MMD2ATL; -- Module Template
2 create OUT : ATL from IN : KM3;
3
4 rule AddedClass {
5   from
6      s : KM3!Class (
7         not s.isAbstract and s.name.startsWith('Added')
8      )
9   using {
10     newHelper : Sequence (ATL!Helper) = OclUndefined;
11     ...
12  }
13  to
14     t : ATL!MatchedRule (
15        name <- s.name +'2'+s.name.regexReplaceAll('Added',''),
16        ...
17  ),
```

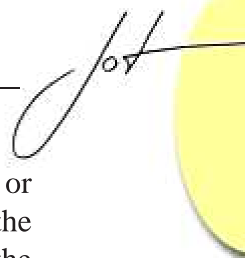
```
18      ...
19    do {
20       ...
21       newHelper<- thisModule.CreateAddedHelper(s);
22       }
23 }
24
25 rule UnchangedClass {
26    from
27      s : KM3!Class (
28         not s.isAbstract and (not s.name.startsWith('Added')) and
29        (not s.name.startsWith('Deleted')) and
30        (not s.name.startsWith('Changed'))
31      )
32    ...
33    to
34      t : ATL!MatchedRule (
35         name <- 'Unchanged'+s.name+'2'+s.name,
36         ...
37       ),
38    ...
39 }
40
41 rule ChangedClass {
42    from
43      s : KM3!Class (
44         not s.isAbstract and s.name.startsWith('Changed')
45      )
46      ...
47    to
48      t : ATL!MatchedRule (
49         name <- s.name +'2'+s.name.regexReplaceAll('Changed',''),
50      ),
51      ...
52 }
```

Listing 3: Fragment of the *MMD2ATL* higher-order transformation

Depending on the structural features of the matched metaclass, a number of helpers (like the one in the line 4 of Listing 2) are created. Since such generations are quite complex and it is difficult to specify them in a declarative way, ATL *called rules* and *action blocks* are used. In particular, a called rule is a rule called by other ones like a procedure. An action block is a sequence of imperative statements and can be used instead of, or in combination with a target pattern in matched or called rules. For instance, lines 19-22 implement an action blocks where the called rule `CreateAddedHelper` is invoked in order to generate the target helpers needed for the management of additions specified in a given model difference (see the lower side of Fig. 8).

5 DIFFERENCE OPERATORS

The evolution of a model consists of the initial model and a number of difference models in such a way the final model is obtained by applying all the modifications to the original one. Starting from a difference model it is useful to automatically generate its dual model, i.e., an inverse difference model that when applied to the final model returns the original one. This allows the designers to operate and store only the final model and eventually rollback to previous versions until the original model. Further useful constructions would be the compositions of delta documents, like sequential and parallel merging of



several versions independently developed. The former can be exploited to group two or more subsequent modifications in a single difference model, while the latter enables the concurrent manipulation of the same artifacts, which will need a fusion step to obtain the overall resulting delta with respect to the previous version.

In the sequel, the mentioned operators will be discussed to suggest how to implement them by means of the proposed approach.

Dual Notion

The dual calculation consists of the following operations: *a)* the `added` and `deleted` specializations are transformed to the corresponding `deleted` and `added` ones, respectively; *b)* the `changed` specialization is moved to the linked element and the direction of the association between them is reversed. The previous steps define a general model transformation which can be easily derived from the source metamodel, as shown in Sect. 3 and Sect. 4 for difference representation and animation, respectively. In essence, a difference model and its dual induce two transformations which are the inverse one with each other.

As mentioned in the previous sections, a difference model is self–contained and minimalistic or, in other words, it must contain all and only the relevant information. This is particularly relevant for the deletion of containers, which requires that all its contained elements are denoted as deleted as well. What appears an unnecessary repetition is indeed a requisite which prevents the dual calculation from navigating the base model: in fact, from a difference model which "deletes" a container, we have to derive a dual model which "add" both the same container and whatever it contains, which is left undetermined without referring to the initial model.

Sequential and Parallel Compositions

As mentioned above, an evolution consists of an initial model and a number of subsequent modification documents. For the sake of simplicity, let us consider only two subsequent modifications over the initial model. The sequential composition of such manipulations corresponds to merge the modifications conveyed by the first document and then, in turn, by the second one in a resulting difference model containing a minimal difference set, i.e., only those modifications which have not been overridden by subsequent modifications. Given a couple of subsequent modifications affecting the same element, the optimization management will behave as summarized in Table 1: when a (`added`, `deleted`) sequence occurs it is possible to ignore both the manipulations being one the dual of the other, while in the case of a (`changed`, `deleted`) it is possible to perform

$\delta_1 \setminus \delta_2$	**added**	**changed**	**deleted**
added	$\setminus$	added $\oplus$ changed	
changed	$\setminus$	changed $\oplus$ changed	deleted
deleted	added	$\setminus$	$\setminus$

Table 1: Optimization cases.

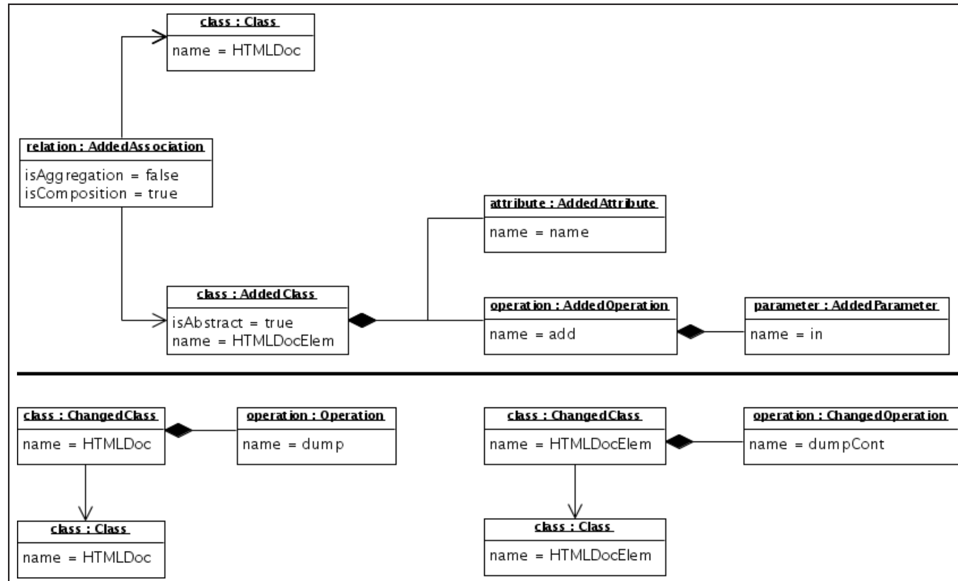
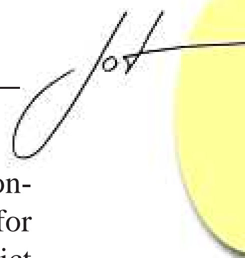


Figure 9: A delta composition example.

only the deletion of the element since the changes would be lost anyway. To compact a (`deleted`, `added`) couple an update should be built which changes the version of the element depicted in `deleted` with the re-added one in `added`. In the situations where a (`added`, `changed`) or a (`changed`, `changed`) occurs, it is possible to group the manipulations in a single `added` and `changed` delta, respectively. In particular, in the former case the addition can be completed with the subsequent changes, whereas in the latter the updates can be composed in a single merged one. Finally, the other couples can be ignored simply because it is not possible they could occur; for instance, it is not possible to update an element before creating it (`changed`, `added`) or to modify a previously deleted element (`deleted`, `changed`). In the work in [22], where the difference example used in this paper has been taken, there is also an intermediate version with respect to the initial and final ones shown here, which can be exploited to illustrate a possible application of composition. In the top of Fig. 9, it is depicted the addition of the `HTMLDocElem` class and the composition relation from `HTMLDoc` class to the just added one. In the lower part of the same figure can be seen two further changes to `HTMLDocElem` and `HTMLDoc` classes. As said above, when an (`added`, `changed`) sequence occurs on the same element, it can be possible to build a single addition completed with the subsequent modifications. Therefore, in the delta document of Fig. 7 it is possible to see a single `added` difference in which the changes contained in the second part have been included.

In a distributed development environment modifications can be operated also diverging from the same ancestor in parallel. In case both modifications are not affecting the same elements (or in other words are parallel independent) their composition is obtained by merging the difference models. This property can be easily shown by performing the parallel independent modifications by interleaving the single changes and assimilating it to the sequential composition. Unfortunately, the result of two parallel modifications can give place to conflicting results, i.e., elements in the original model which are changed



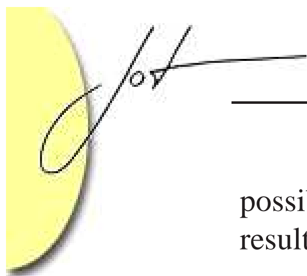
by both difference models without converging to a common result. In this case, conflicting modifications either have to be resolved by the corresponding designers (see for instance [1]), or they need some mechanism to support such a task [8]. However, conflict resolution is a current research topic but it goes far beyond the scope of this paper.

6 RELATED WORK

This paper is related to numerous aspects of modeling, even if only few of them are pertinent to difference representation. In fact, there exist methods and algorithms for detecting differences between documents (see [2, 26, 27, 30] for UML–aware calculations and [20] for a metamodel independent approach, respectively) and only few focus on visualization issues. As discussed in Sect. 2, difference representations are usually given by means of edit scripts or coloring techniques. These formalizations present limitations since their lack of abstraction and declarativeness prevents them from being represented by suitable metamodels and processed in standard modeling platforms. In [6] the problem of representing changes between data structures in EMF Service Data Objects (SDO) is addressed. Change summaries are used for recording modifications within data graphs which consist mainly of trees of data objects; data graphs are serialized to XML and change summaries make use of XPath for querying documents. With respect to our proposal, change summaries do not meet the requirements of model-based, metamodel independence, and transformability.

Although our approach is agnostic of the calculation method, it can be interesting to consider the way certain difference calculations are performed. In fact, some UML–differencing techniques are based on persistent identifiers which are assigned to all model elements. This characteristic locks the designer within a tool since models realized with different modeling tools have different identifiers and are therefore not comparable. A solution to this problem is introduced in [29] where algorithms based on similarity analysis are able to detect changes without referring to persistent identifier. This approach is based on the notion of longest common subsequence. The presentation is usually realized by means of change tree visualization introduced in two different versions which are essentially the same but follow the inheritance– and containment–spanning tree of software model, respectively. Such representation requires ad–hoc tool support. Persistent identifiers could have been employed also in difference representation avoiding to have deep copies of model elements and keeping models relatively small in size. Unfortunately, using persistent identifiers poses a number of questions, in particular: *a)* it tends to lock a model life-cycle within a specific tool; *b)* it reduces the applicability of the induced transformation since models edited independently (even within the same tool) from the base model have different identifiers although conceptually related; *c)* finally, it would likely restrict the adoptable calculation methods to those methods based on persistent identifiers.

An approach to compare models which conform to arbitrary metamodels is proposed in [20]; difference calculation is performed by means of a similarity analysis technique, while the representation exploits a coloring method based on tree arrangement of the detected structural changes. From our point of view that work can be considered as a



possible candidate technique to perform a metamodel independent differentiation whose results can be represented with similar techniques as those presented in this paper.

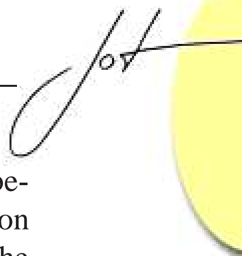
An interesting initiative related to this work is the FAMOOS project [11], whose goal is to build a framework to support the evolution and reengineering of object–oriented software systems. In particular, the language–independent FAMIX metamodel [28] is often used for modelling snapshots in approaches which handle the history as a first class entity, as for instance in [15] where the Hismo metamodel is introduced. In our approach subsequent versions are represented by models which are possibly related via difference models. In contrast with Hismo, we cannot explicitly model history and versions. Thus, as far as we know, it seems that the approaches are somewhat orthogonal since we are able to specify arbitrary snapshots which are linked by automated transformations induced by the modifications.

Darcs [25] is a text-based revision control system (not dissimilar to CVS) which is based on a theory of patches, whose properties enable context-independent manipulations. Several concepts and issues in this paper have analogies with such theory. In particular, every patch is required to be invertible. Sequential composition of patches may be subject to a reordering which can fail because of missing dependencies. Moreover, concurrent patches (i.e., patches that are applied on the same source tree) can be merged; the result of a set of merges is independent of the order in which the merges are performed. Darcs can be a relevant source of inspiration for our work since there is at least a perfect analogy across the corresponding domains.

7 CONCLUSIONS AND FUTURE WORK

This paper discussed the problem of representing differences among models conforming to an arbitrary metamodel. Differences can be therefore given as a model which adheres to a difference metamodel obtained by an automated transformation. Interestingly, difference models, regardless of the metamodel the base models are conformant to, are given a behavior which transforms an initial model to the final one by means of a higher-order transformation. The proposal has been devised to comply to the “everything is a model” principle and to be accommodated in a generic modeling framework as shown by the implementation [9] upon the AMMA framework. Compositional operators which combine models sequentially and in parallel have been introduced.

Future work will address the problem of conflict detection and resolution. Parallel modifications can give place to conflicts which are usually detected by means of traditional lexical approaches which lack of abstraction and can give place to false positive and negative issues. Our goal is to define a weaving metamodel for the specification of conflicts in such a way designers can *customize* their notion of conflict according to the specific stage in the development process they are dealing with. A preliminary investigation has been recently presented in [8]. Model differences present an analogy to the patch utility. Since the induced transformations do not present any adjustability of their application, we intend to investigate how to introduce a fuzziness factor. In particular, we plan to



adopt weaving models [3] for setting – at different level of granularity – relationships between model elements and modifications to be applied to. Furthermore, the composition of difference models has been partly investigated in [8], thus the implementation of the corresponding higher-order transformations has not been totally realized and it is planned to be covered in the near future.

Acknowledgments. This work has been partially supported by the IST EU project "PLASTIC" (www.ist-plastic.org).

REFERENCES

[1] M. Alanen and I. Porres. Difference and Union of Models. In *Procs. UML 2003*, volume 2863 of *LNCS*, pages 2–17.

[2] D. T. Barnard, G. Clarke, and N. Duncan. Tree-to-tree correction for document trees. Technical report, Departement of Computing and Information Science Queen's University Kingston, Canada, Jan 1995.

[3] J. Bézivin. On the Unification Power of Models. *SOSYM*, 4(2):171–188, 2005.

[4] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In *Model Driven Architecture, ECMDA Workshops: Foundations and Applications*, volume 3599 of *LNCS*, pages 33–46, 2004.

[5] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T.J. Grose. *Eclipse Modeling Framework*. Addison Wesley, 2003.

[6] M. Carey. Data Delivery in a Service Oriented World: The BEA AquaLogic Data Services Platform.

[7] S.-Y. Chien, V. J. Tsotras, and C. Zaniolo. *Version Management of XML Documents*, volume 1997 of *LNCS*, pages 184–200. Springer-Verlag.

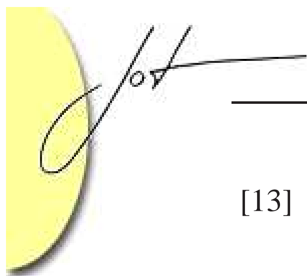
[8] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Composition of Model Differences. In *Procs. CMT 2006*, number TR-CTIT-06-34 in CTIT Technical Reports.

[9] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. ATL Use Case - A Metamodel Independent Approach to Difference Representation, 2007. http://www.eclipse.org/m2m/atl/usecases/MMIndApproachtoDiffRep/.

[10] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[11] S. Ducasse and S. Demeyer. The FAMOOS Object-Oriented Reengineering Handbook.

[12] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Trans. Software Eng.*, 27(1):1–12, 2001.



[13] S. G. Eick, J. L. Steffen, and E. E. Sumner Jr. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Software Eng.*, 18(11):957–968, 1992.

[14] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Procs. ICSM 2003*, pages 23–32. IEEE Computer Society.

[15] T. Girba and S. Ducasse. Modeling History to Analyze Software Evolution. *J. Softw. Maint. Evol.: Res, Pract.*, 18:207–236, 2006.

[16] F. Jouault and J. Bézivin. KM3: a DSL for Metamodel Specification. In *Procs. FMOODS'06*, volume 4037 of *LNCS*, pages 171–185.

[17] F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *Procs. Fifth Intl. Conference on Generative Programming and Component Engineering (GPCE'06)*, 2006. to appear.

[18] F. Jouault and I. Kurtev. Transforming Models with ATL. In *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 128–138. Springer-Verlag, 2005.

[19] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. In *Workshop on Intelligent Signal Processing*, 2001.

[20] Y. Lin, J. Gray, and F. Jouault. DSMDiff: A Differentiation Tool for Domain-Specific Models, Dec 2006.

[21] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.

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

[23] OMG. *Meta Object Facility (MOF) 2.0 Core Specification, OMG Document ptc/03-10-04*. http://www.omg.org/docs/ptc/03-10-04.pdf, 2003.

[24] OMG. OCL 2.0 Specification, 2006. OMG Document formal/2006-05-01.

[25] D. Roundy. Darcs 1.0.9rc2 Official Manual.

[26] S. M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, 1977.

[27] K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, 1979.

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

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

[30] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245–1262, 1989.

ABOUT THE AUTHORS

Antonio Cicchetti is a Ph.D student in the Computer Science Department at the University of L'Aquila, Italy from 2004. His research interests include techniques for model differencing and management in current model-engineering platforms, domain-specific modelling languages, model transformations and model weaving. He can be reached at cicchetti@di.univaq.it. See also http://www.di.univaq.it/∼cicchetti.

Davide Di Ruscio recently has received his Ph.D in Computer Science from the University of L'Aquila, Italy. His research interests include generative techniques and methodologies for Web development, model driven engineering and more specifically model transformation and model differencing. He can be reached at diruscio@di.univaq.it. See also http://www.di.univaq.it/∼diruscio.

Prof. Alfonso Pierantonio is Associate Professor in the Computer Science Department at the University of L'Aquila, Italy. His present research interests include general model engineering and more specifically model transformation and techniques for model differencing and management in current model-engineering platforms. He has been involved in program and organization committees of conferences and co-edited several special issues on scientific journals about these subjects. He can be reached at alfonso@di.univaq.it. See also http://www.di.univaq.it/∼alfonso.