Representing and Operating with Model Differences

José E. Rivera and Antonio Vallecillo

Dept. Lenguajes y Ciencias de la Computación
ETS Ingeniería Informática
Universidad de Málaga, Spain
{rivera,av}@lcc.uma.es

Abstract. Models and metamodels play a cornerstone role in Model-Driven Software Development (MDSD). Models conform to metamodels, which usually specify domain-specific languages that allow to represent the various facets of a system in terms of models. This paper discusses the problem of calculating differences between models conforming to arbitrary metamodels, something essential in any MDSD environment for dealing with the management of changes and evolution of software models. We present a metamodel for representing the differences as models, too, following the MDSD "everything is a model" principle. The *Difference Metamodel*, together with the difference and other related operations (`do`, `undo` and composition) presented here have been specified in Maude and integrated in an Eclipse-developed environment.

Keywords: Model-driven software development, model difference, model comparison, model evolution, Maude, object matching.

1 Introduction

Model-Driven Software Development (MDSD) is becoming a widely accepted approach for developing complex distributed applications. MDSD advocates the use of models as the key artifacts in all phases of development, from system specification and analysis, to design and implementation. Each model usually addresses one concern, independently from the rest of the issues involved in the construction of the system.

Domain-Specific Modeling (DSM) is a way of designing and developing systems that involves the systematic use of Domain Specific Languages (DSLs) to represent the various facets of a system, in terms of models. Such languages tend to support higher-level abstractions than general-purpose modeling languages, and are closer to the problem domain than to the implementation domain. Thus, a DSL follows the domain abstractions and semantics, allowing modelers to perceive themselves as working directly with domain concepts. Furthermore, the rules of the domain can be included into the language as constraints, disallowing the specification of illegal or incorrect models. The abstract syntax of a DSL is usually described by a metamodel.

So far, most of the efforts have been focused on the definition of models, metamodels and transformations between them. Nowadays, other operations such as

R.F. Paige and B. Meyer (Eds.): TOOLS EUROPE 2008, LNBIP 11, pp. 141–160, 2008.

model subtyping [1], type inference [2] and model difference [3] are becoming increasingly important, too, in order to provide full support to Model-driven Engineering practices.

In particular, model difference is an essential operation in several software development processes [4], including version and change management, software evolution, model/data integration, etc. At the moment, techniques for visualizing and representing model differences are mainly based on edit scripts and coloring techniques [5,6,7]. However, these approaches do not produce models as results of their calculations, and therefore cannot be fully integrated with other MDSD processes. Furthermore, most of them do not fulfill other interesting properties required in MDSD environments, such as composability [8]. Other techniques based on text, data structure or models also exist but are usually restricted to a specific metamodel (namely UML) [5,9,10]. In this paper we present an approach to compare models which conform to arbitrary metamodels. For this purpose we have defined a *Difference Metamodel* so that differences are represented as models, too, that conform to such a metamodel. We have also defined a set of operations on models and on differences that provide support for the calculation of differences, their application and composition. The *Difference Metamodel* and those difference operations have been specified in Maude using the formal notation proposed in [2] to represent models and metamodels, and integrated in our Eclipse developed environment called Maudeling [11].

There are several reasons that moved us to formalize our definitions and specifications in Maude. Firstly, in this way we can provide precise definitions of the concepts and operations, at a high level of abstraction, and independently from the particularities of any implementation programming language (such as Java, Python, etc.). Secondly, having formal descriptions also allows the analysis of the specifications produced. Finally, the fact that Maude specifications are executable (with comparable performance to most commercial programming languages) has permitted us to count on efficient implementations of the concepts and operations described here, which are correct by construction.

The structure of this document is as follows. First, Sections 2 and 3 provide a brief introduction to Maude, and how models and metamodels can be represented in Maude, respectively. Section 4 presents our definition and specification of the model difference operation, and the *Difference Metamodel* in which the results are expressed. Section 5 introduces some other difference related operations, their specification in Maude, and the supporting tool. Section 6 compares our work with other related proposals. Finally, Section 7 draws some conclusions and outlines some future research activities.

2 Rewriting Logic and Maude

2.1 Introduction to Maude

Maude [12,13] is a high-level language and a high-performance interpreter and compiler in the OBJ algebraic specification family. It supports membership-equational logic and rewriting logic specification and programming of systems.

Thus, Maude integrates an equational style of functional programming with rewriting logic computation. Because of its efficient rewriting engine, able to execute more than 3 million rewriting steps per second on standard PCs, and because of its metalanguage capabilities, Maude turns out to be an excellent tool to create executable environments for various logics, models of computation, theorem provers, or even programming languages. In addition, Maude has been successfully used in software engineering tools and several applications [14]. We informally describe in this section those Maude's features necessary for understanding the paper; the interested reader is referred to its manual [13] for more details.

Rewriting logic is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. A distributed system is axiomatized in rewriting logic by an equational theory describing its set of *states* and a collection of rewrite rules. Maude's underlying equational logic is membership-equational logic, a Horn logic whose atomic sentences are equalities $t = t'$ and *membership assertions* of the form $t : S$, stating that a term t has sort S.

Computation in a functional module is accomplished by using the equations as simplification rules from left to right until a canonical form is found. Some equations, like those expressing the commutativity of binary operators, are not terminating but nonetheless they are supported by means of *operator attributes*, so that Maude performs simplification modulo the equational theories provided by such attributes, which can be associativity (`assoc`), commutativity (`comm`), identity (`id`), and idempotence (`idem`).

While functional modules specify membership-equational theories, rewrite theories are specified by *system modules*. A system module may have the same declarations of a functional module plus rules of the form $t \rightarrow t'$, where t and t' are terms. These rules specify the dynamics of a system in rewriting logic. They describe the local, concurrent transitions possible in the system, i.e., when a part of the system state fits the pattern t then it can change to a new local state fitting pattern t'. The guards of conditional rules act as blocking pre-conditions, in the sense that a conditional rule can only be fired if the condition is satisfied.

2.2 Object-Oriented Specifications: Full Maude

In Maude, concurrent object-oriented systems are specified by object-oriented modules in which classes and subclasses are declared. A class is declared with the syntax `class` C `|` $a_1 : S_1$ `,` ... `,` $a_n : S_n$, where C is the name of the class, a_i are attribute identifiers, and S_i are the sorts of the corresponding attributes. Objects of a class C are then record-like structures of the form `<` $O : C$ `|` $a_1 : v_1$ `,` ... `,` $a_n : v_n$ `>`, where O is the name of the object, and v_i are the current values of its attributes. Objects can interact in a number of different ways, including message passing. Messages are declared in Maude in `msg` clauses, in which the syntax and arguments of the messages are defined.

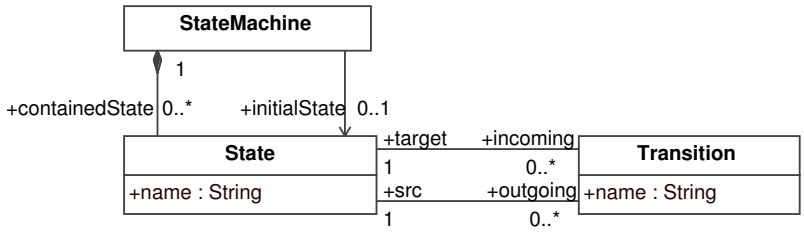

Fig. 1. Simple State Machine Metamodel

In a concurrent object-oriented system, the concurrent state, which is called a *configuration*, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting using rules that describe the effects of the communication events of objects and messages. The predefined sort `Configuration` represents configurations of Maude objects and messages, with `none` as empty configuration and the empty syntax operator `__` as union of configurations.

Class inheritance is directly supported by Maude's order-sorted type structure. A subclass declaration `C < C'`, indicating that `C` is a subclass of `C'`, is a particular case of a subsort declaration `C < C'`, by which all attributes, messages, and rules of the superclasses, as well as the newly defined attributes, messages and rules of the subclass characterize its structure and behavior. This corresponds to the traditional notion of subtyping: A is a subtype of B if every <X> that satisfies A also satisfies B. Multiple inheritance is also supported in Maude [12,15].

3 Formalizing Models and Metamodels with Maude

There are several notations to represent models and metamodels, from textual to graphical. In [2] we presented a proposal based on the use of Maude, which not only was expressive enough for these purposes, but also offered good tool support for reasoning about models. In particular, we showed how some basic operations on models, such as model subtyping, type inference, and metric evaluation, can be easily specified in Maude, and made available in development environments such as Eclipse. This section presents just a brief summary of that proposal.

In Maude, models are represented by configurations of objects. Nodes are represented by Maude objects. Nodes may have attributes, that are represented by Maude objects' attributes. Edges are represented by Maude objects' attributes, too, each one representing the reference to the target node of the edge.

Then, metamodels are represented by Maude object-oriented modules. They contain the specification of the Maude classes to which the Maude objects (that represent the corresponding models nodes) belong. In this way, models conform to metamodels by construction.

To illustrate this approach, the following piece of Maude specifications describe a Simple State Machine metamodel (depicted in Fig. 1) as a Maude module.

```
(omod SimpleStateMachines is
   protecting STRING .
   class State |
       name : String,
       stateMachine : Oid,
       incoming : Set{Oid},
       outgoing : Set{Oid} .
   class StateMachine |
       containedStates : Set{Oid},
       initialState : Maybe{Oid} .
   class Transition |
       name : String,
       target : Oid,
       src : Oid .
endom)
```

Metaclasses correspond to Maude classes. Meta-attributes are represented as Maude attributes. Meta-references are represented as attributes too, by means of sets of Maude object identifiers. Depending on the multiplicity, we can use: a single identifier (if the multiplicity is `1`); a `Maybe{Oid}` which is either an identifier or a `null` value, for representing a `[0-1]` multiplicity; a `Set{Oid}` for multiplicity `[*]`; or a `List{Oid}` in case the references are `ordered`.

The instances of such classes will represent models that conform to the example metamodel. For instance, the following configuration of Maude objects shows a possible state machine model that conforms to the `SimpleStateMachines` metamodel:

```
< 'SM : StateMachine | initialState : 'ST1,
   containedStates : ('ST1, 'ST2) >
< 'ST1 : State | name : "St1", stateMachine : 'SM,
   outgoing : 'TR, incoming : empty >
< 'ST2 : State | name : "St2", stateMachine : 'SM,
   incoming : 'TR, outgoing : empty >
< 'TR : Transition | name : "Tr", src : 'ST1, target : 'ST2 >
```

It represents a simple state machine with two states, named `St1` and `St2`, and one transition (`Tr`) between them. `St1` is the initial state of the state machine.

The validity of the objects in a configuration is checked by the Maude type system. In addition, other metamodel properties, such as the valid types of the object referenced, or the valid opposite of a reference (to represent bidirectional relationships), are expressed in Maude in terms of membership axioms. Thus, membership axioms will define the well-formedness rules that any valid model should conform to: a configuration is valid if it is made of valid objects, with valid attributes and references. The well-formedness rules of the simple state machines metamodel can be found in [2].

For those users familiar with the Ecore terminology, there is no need to define the metamodel initially in Maude: ATL (Atlas Transformation Language) model transformations have been defined from Ecore to Maude specifications.

Finally, note that since metamodels are models too, they can also be represented by configurations of objects. The classes of such objects will be the ones specified in the metametamodels, for example, the classes that define the MOF or Ecore metamodels. In this way, metamodels can be handled in the same way as models are, i.e., operations defined over models can be applied to metamodels as well.

4 Model Difference

Having described how models and metamodels can be represented in Maude, this section introduces a model difference definition and its specification in Maude. Thus, both a metamodel to represent model differences and operations to calculate and operate with them are presented.

4.1 Representation: The Difference Metamodel

Our first requirement is that the results of a model difference operation can be expressed as a model, so they can be fully integrated into other MDSD processes. Since models conform to metamodels, we have to define a *Difference Metamodel* with the elements that a difference may contain, and the relationships between them. Furthermore, the *Difference Metamodel* should be general enough to be independent of the metamodel of the source models.

Taking into account these characteristics, we have developed the *Difference Metamodel*, which is depicted in Fig. 2. A difference model will contain all the changes from a *subtrahend* model to a *minuend* model. As usual, we can distinguish three different kinds of changes: element addition, element deletion and element modification. Thus, every element of a difference model (`DiffElement`) will belong to `ModifiedElement` metaclass, `DeletedElement` metaclass or `AddedElement` metaclass, depending on whether the element has been added, deleted or modified, respectively. Elements which do not suffer from any changes, will not be reflected in the difference model.

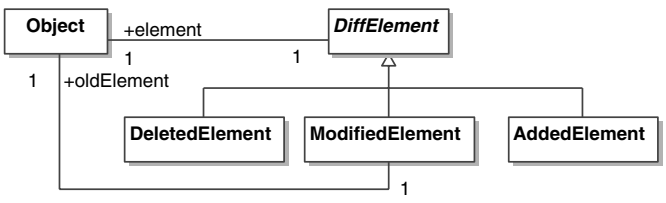

Fig. 2. Difference Metamodel

Every difference element `DiffElement` will have a reference (`element`) to the element that has suffered the change. In case of element modification (`ModifiedElement`), the difference element will refer to both the element of the minuend model (after the modification, `element`), and the element of the subtrahend model (before the modification, `oldElement`).

Modified, deleted and added elements from both operand models are added to the difference model too, so that it is self-contained [8], i.e., the difference model will contain all the changes, not relying on external sources of information (such as the operand models). These elements can belong to any metaclass, since model difference can be applied to models conforming to arbitrary metamodels. Thus we introduce the `Object` metaclass (similar to *EObject* in Ecore) to represent such arbitrary metaclasses.

The *Difference Metamodel* can be specified in Maude as follows:

```
(omod ModelDiff is
  class Object .
  class DiffElement | element : Oid .
  class AddedElement .
  class DeletedElement .
  class ModifiedElement | oldElement : Oid .
  subclasses AddedElement DeletedElement ModifiedElement < DiffElement .
 endom)
```

As a matter of fact, class *Object* would not need to be explicitly specified in Maude, because class `Cid` (class identifier) is defined in Maude for this purpose [15]. However, we have defined it for understandability reasons.

4.2 Specification of the Difference Operation

Given a minuend model M_m and a subtrahend model M_s, both conforming to some metamodels (not necessary the same, as we shall later see), the result of applying the model difference operation to them is another model M_d conforming to the *Difference Metamodel* presented above, in such a way that `modelDiff(`M_m`,`M_s`) =` M_d.

The global comparison process is generally admitted as being composed of two main parts: matching and differencing. The latter makes use of the former to decide whether an element in the minuend model is the *same* (although possibly modified) as another in the subtrahend model. Decomposing the difference operation in these two parts allows the reuse of both algorithms in different applications, such as model *patching* [8]. Thus, we will firstly show how elements are matched, and secondly how the the difference is computed using this information.

Matching Elements. Matching two models M_1 and M_2 conforming to some metamodels (not necessary the same) means finding different objects from both models that represent the same element. The result of applying the match operation to M_1 and M_2 is a match model M_M conforming to the *Match Metamodel*, depicted in Fig. 3.

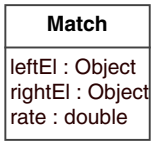

Fig. 3. Match Metamodel

Match model elements (of class `Match`) symbolize links between two objects that represent the same element. Thus, a match model element will refer to both objects (`leftEl` and `rightEl`) and will `rate` their similarity (expressed in terms of a ratio between zero and one).

Matching objects using persistent identifiers. Since Maude objects have persistent identifiers, checking whether two objects represent the same element can be easily done by comparing their identifiers. If their identifiers are the same, the two objects can be said to represent the same element; otherwise, the two objects represent different elements.

Given variables `O`, `O1` and `O2` of sort `Oid`; `C1` and `C2` of sort `Cid`; `ATTS1` and `ATTS2` of sort `AttributeSet`; and `CONF1` and `CONF2` of sort `Configuration`, the `match` operation can be specified in Maude as follows:

```
subsort MatchModel < Configuration .
op match : Configuration Configuration -> MatchModel .
eq match(< O : C1 | ATTS1 > CONF1, < O : C2 | ATTS2 > CONF2)
   = < O : Match | leftEl : O, rightEl : O, rate : 1.0 >
     match(CONF1, CONF2) .
eq match(CONF1, CONF2) = none [owise] .
```

For every object of M_1 with the same identifier as another object of M_2, a *match object* that relates them is added to the resulting model. Since `[owise]` equations are only executed if no other equation holds, when no objects with the same identifier are found (Maude provides configuration matching and reorganization facilities) no more information is added to the match model.

Matching objects using structural similarities. Using persistent universal identifiers makes the matching process simple and robust. However, counting on this kind of identifiers is not always possible: if the two models to compare conform to different metamodels, or have evolved independently, there is little chance that an element being the "same" in the two models has the same identifier. In addition, when comparing models not originally specified in Maude but, e.g., in MOF or Ecore, we cannot assume that the model transformations from MOF or Ecore to Maude will assign the same identifier to two different objects that represent the same element.

A more sophisticated matching algorithm is thus needed. This kind of matching algorithm should compare two elements by their structural similarities. There are several structural matching algorithms described in the literature that can be used, e.g. [16,17]. One of the advantages of using Maude is that this kind of

algorithms are usually very easy to specify (compared to those specified in, e.g., Java) thanks to the configuration matching and reorganization facilities that Maude provides.

In this paper we present the structural matching algorithm used in our Maudeling framework [11]. The algorithm starts by comparing every object of a model with every object of the other one. Comparing two objects means comparing their metaclasses and structural features to obtain a final joint `rate`. This match rate will represent the similarity between the two compared objects, which are said to *potentially match* when the rate is greater than a given threshold (Th). At the end of the process, a sieve is applied to all potential matches in order to pair only those objects that together obtain the biggest rate, taking into account that a specific object can only belong to one match relationship.

Class and structural features match rates are obtained in the following way:

- Two metaclasses match if they are the same, or there exists an inheritance relation between them.

$$classRate(C_1, C_2) = \begin{cases} 1.0 & \text{if } C_1 = C_2 \\ 0.9 & \text{if } isSubtype(C_1, C_2) \text{ or } isSubtype(C_2, C_1) \\ 0.0 & \text{otherwise} \end{cases}$$

- Structural features are compared, and given a weight, depending on its type. To obtain the final structural features rate, every attribute rate and weight are jointly considered: $sfRate((S_1, S_2, ..S_n), (R_1, R_2, ..R_n)) = w_1 * rate(S_1, R_1) + w_2 * rate(S_2, R_2) + ... + w_n * rate(S_n, R_n)$.
  If a structural feature's upper cardinality is greater than 1 (i.e., if its value is a collection), the average rate is calculated. If a structural feature is defined only in one of the objects, a penalty is applied to the final $sfRate$.
  - Boolean attributes and enumerations match (with $rate = 1.0$) if they have the same value (otherwise $rate = 0.0$).
  - String attribute values distances are calculated using the Levenshtein algorithm [18]. The Levenshtein distance is the minimum number of operations (insertion, deletion, or substitution of a single character) needed to transform one string into another. Depending on the resulting distance, a different rate is given.

$$nameRate(S_1, S_2) = \begin{cases} 1.0 & \text{if } levenshteinDist(S_1, S_2) = 0 \\ 0.9 & \text{if } levenshteinDist(S_1, S_2) = 1 \\ 0.5 & \text{if } levenshteinDist(S_1, S_2) = 2 \\ 0.1 & \text{if } levenshteinDist(S_1, S_2) = 3 \\ 0.0 & \text{otherwise} \end{cases}$$

  - Numerical attribute values match rate is computed with a relative distance function ($1 - \frac{|N_1 - N_2|}{|N_1 + N_2|}$ limited to [0..1])
  - References are matched recursively, i.e., objects referenced are compared using the same match operation but without taking into account their own references (to avoid cycles).

Once the class and structural features match rates are calculated, the final joint `rate` is obtained as follows:

$$finalRate = w_c * classRate + w_{sf} * sfRate + w_n * nameRate$$

where $finalRate, classRate, nameRate, sfRate \in [0..1]$, and the weights that we have initially considered are $w_c = 0.5, w_{sf} = 0.25, w_n = 0.0$. The threshold value we normally use is $Th = 0.66$, although the weights and threshold are of course user-defined and easily configurable.

It is worth noting that the weights and threshold values specified above do not allow elements to potentially match if their metaclasses are not related. In addition, a *nameRate* has been included in the equation. In many kinds of models the attribute `name` is considered as an identifier. By including this rate we allow elements of this kind of models to be better matched (assigning the attribute `name` a bigger weight than any other structural feature). For instance, setting the name weight to $w_n = 0.25$, will make objects with the same class and same name to always potentially match. Attribute `name` values are compared in the same way as string values, i.e., using the Levenshtein distance.

If the name rate is omitted (i.e., it is not considered in the computations: $w_n = 0.0$), potential matches are harder: structural features should be more strongly related because no identifier is provided. In all cases, i.e., either using the name as an identifier or not, renamed elements can be detected (objects with different name can potentially match).

Contrary to other approaches (e.g., [16]) in which a model is seen as a tree (levels are determined by the containment relationship), and only objects at the same level are compared, our approach compares every object independently of its depth in the tree. This decision implies more comparisons, but also brings along interesting advantages: (a) moved elements through different levels can be detected; and (b) failing to identify a match does not condition other potential matches below in the tree hierarchy. For example, refactoring is a common technique used for making models evolve. One usual refactorization step is to add packages to improve the grouping structure of the model. This is the kind of change that affects the containment tree, and that can be missed by those approaches that compare elements only at the same level of the tree.

Specifying the Calculation of Differences. As previously mentioned, the model difference operation makes use of the match model in order to decide whether one element in the minuend model is the *same* (although possibly modified) as another in the subtrahend model. Thus, in the global comparison process the match model is calculated before the differencing part starts:

```
subsort DiffModel < Configuration .
op modelDiff : Configuration Configuration -> DiffModel .
op modelDiff : Configuration Configuration MatchModel -> DiffModel .
eq modelDiff(CONF1, CONF2) =
   modelDiff(CONF1, CONF2, match(CONF1,CONF2)) .
```

In order to specify the behavior of the differencing part, we have identified four different situations that may happen when calculating a model difference

operation on an element: (1) the element appears in both models (minuend and subtrahend) and has not been modified; (2) the element appears in both models but has been modified; (3) the element only appears in the minuend model; (4) the element only appears in the subtrahend model.

The following four Maude equations specify the `modelDiff` operation in each case. In all of them we will use the following variables: `O`, `O1` and `O2` of sort `Oid`; `C`, `C1` and `C2` of sort`Cid`; `ATTS`, `ATTS1` and `ATTS2` of sort `AttributeSet`; and `CONF`, `CONF1`, `CONF2` and `MATCHM` of sort `Configuration`.

In the first case, we have to check whether two objects (one belonging to the minuend model, the other belonging to the subtrahend model) match, i.e., they represent the same element, and belong to the same class and have the same attribute values (remember that in Maude both object attributes and references are expressed by Maude attributes). If this situation occurs, we have found an element that has not been modified, and therefore no evidence of the element is stored in the difference model:

```
ceq modelDiff(< O1 : C | ATTS > CONF1> < O2 : C | ATTS > CONF2, MATCHM)
  = modelDiff(CONF1, CONF2, MATCHM)
  if match(O1, O2, MATCHM) .
```

The match operation checks whether the corresponding match object that relates `O1` and `O2` exists in the match model.

```
op match : Oid Oid Configuration -> Bool .
eq match(O1, O2, < O : Match | leftEl : O1, rightEL : O2, ATTS > CONF)
   = true .
eq match(O1, O2, CONF) = false [owise] .
```

In the second case, two objects represent the same element, but the element has been modified, i.e., the two objects match, but either they belong to different classes (Maude allows the dynamic reclassification of objects), or their attributes have different values. In this case, we create an object instance of class `ModifiedElement` with references to both the object of the subtrahend model (before the modification, `oldelement`) and the object of the minuend model (after the modification, `element`). Both operand models' objects are added to the difference model, but only with the relevant attributes, i.e., those that have different values in both objects (storing both values, the old one and the new one), or those that are specified in one object but not in the other (corresponding to deleted attributes if the attributes are specified in objects of the subtrahend model, or corresponding to added attributes if they are specified in objects of the minuend model). The identifiers of the two added objects are modified (with `newId` and `oldId` operations) to distinguish them, since Maude objects should have unique identifiers in the same Maude configuration.

Modifications to object identifiers are performed in such a way that it would be possible to "undo" them to get the original identifiers (with `originalId` operation, defined in next section).

```
ceq modelDiff( < O1 : C1 | ATTS1 > CONF1,
               < O2 : C2 | ATTS2 > CONF2, MATCHM)
  = < newModId(O1) : ModifiedElement |
      element : newId(O1), oldElement : oldId(O2) >
    < newId(O1) : C1 | attsDiff(ATTS1,ATTS2) >
    < oldId(O2) : C2 | attsDiff(ATTS2,ATTS1) >
    modelDiff(CONF1,CONF2,MATCHM)
 if match(O1, O2, MATCHM) /\ (not(ATTS1 == ATTS2) or not(C1 == C2)) .
```

Note that every element modification is treated in the same way, i.e., metaclass `ModifiedElement` is used for all kinds of feasible modifications: from a modification in a `String` attribute value to a change in the order of elements in collections. This decision was made for the sake of simplicity although, of course, the *Difference Metamodel* could be easily extended to explicitly distinguish between different kinds of element modifications, if required.

In the third and fourth cases, one element in one of the models does not match any other element of the other model. If the object only appears in the minuend model, the element has been added; otherwise (i.e., the object only appears in the subtrahend model) the element has been deleted. Thus, we just have to create an object `AddedElement` (or `DeletedElement`, respectively), with a reference to the element in question which will be also added to the difference model (modifying its identifier as previously described):

```
eq modelDiff( < O1 : C1 | ATTS1 > CONF1, CONF2, MATCHM )
  = < newAddId(O1) : AddedElement | element : newId(O1) >
    < newId(O1) : C1 | ATTS1 >
    modelDiff(CONF1, CONF2, MATCHM) [owise] .

eq modelDiff( CONF1, < O2 : C2 | ATTS2 > CONF2, MATCHM )
  = < newDelId(O2) : DeletedElement | element : oldId(O2) >
    < oldId(O2) : C2 | ATTS2 >
    modelDiff(CONF1, CONF2, MATCHM) [owise] .
```

Finally, the reader should notice the existence of a final fifth case in case both the minuend and subtrahend models are empty. The result of the `modelDiff` operation will be an empty difference model, as expected:

```
eq modelDiff( none, none, MATCHMODEL ) = none .
```

4.3 An Example

For illustration purposes, let us introduce a simple example to show how the model difference works, and the results that it obtains. Given the state machine model presented in Section 3, suppose that we add a transition in the opposite direction to the existing one, i.e., a new transition `Tr2` from state `St2` to state `St1`. As a result, the following model is obtained:

```
< 'SM : StateMachine | initialState : 'ST1 ,
   containedStates : ('ST1, 'ST2) >
< 'ST1 : State | name : "St1", stateMachine : 'SM,
   outgoing : 'TR, incoming : 'TR2 >
< 'ST2 : State | name : "St2", stateMachine : 'SM,
   outgoing : 'TR2, incoming : 'TR2 >
< 'TR : Transition | name : "Tr", src : 'ST1 , target : 'ST2 >
< 'TR2 : Transition | name : "Tr2", src : 'ST2 , target : 'ST1 >
```

Note that states `St1` and `St2` are also modified since they have a reference to the incoming and outgoing transitions.

Now, if we take the modified model as the minuend model, and the initial model as the subtrahend model, the result of applying the difference operation is a model (shown below) that conforms to the *Difference Metamodel*:

```
< 'ST1@MOD : ModifiedElement | element : 'ST1@NEW,
  oldElement : 'ST1@OLD>
< 'ST1@NEW : State | incoming : 'TR2 >
< 'ST1@OLD : State | incoming : empty >
< 'ST2@MOD : ModifiedElement | element : 'ST2@NEW,
  oldElement : 'ST2@OLD >
< 'ST2@NEW : State | outgoing : 'TR2 >
< 'ST2@OLD : State | outgoing : empty >
< 'TR2@ADD : AddedElement | element : 'TR2@NEW >
< 'TR2@NEW : Transition | name : "Tr2", src : 'ST2, target : 'ST1 >
```

As we can see, both added transition and states reference modifications are represented in the difference model. Elements were matched as expected, since their name and class were not modified (with $w_n = 0.25$).

An interesting property of this difference operation is that it can be applied to minuend and subtrahend models conforming to different metamodels. Thus, in some situations in which models and metamodels are evolving at the same time, models can be compared as well. Every element (and attribute value) is handled in the same way, no matter whether its metaclass (or any of its meta-attributes) is only defined in the metamodel of one of the (subtrahend or minuend) models.

5 Further Operations

Model difference is probably the main operation for dealing with model evolution and for handling model versions, but it is not the only one required to achieve such processes. There are other related operations that need to be considered too, such as those that `do` and `undo` the changes, `compose` several differences, etc. For instance, operations `do` and `undo` will allow us to obtain the minuend model from the subtrahend model, and viceversa, respectively.

In fact, one of the current limitations of other proposals that implement model comparison and difference (e.g. [7]) is that their results cannot be effectively composed, and that these additional operations are hard to define. In our approach, given the way in which the differences have been represented (as models), and

the `modelDiff` operation has been specified (as an operation on models), they become natural and easy to define.

5.1 The "do" Operation

Given a model M_s conforming to an arbitrary metamodel MM and a difference model M_d (conforming to the *Difference Metamodel*), the result of applying operation `do` to them is another model M_m so that: `do(`M_s`,`M_d`) =` M_m and `modelDiff(`M_m`,`M_s`) =` M_d.

Operation `do` applies to a model all the changes specified in a difference model. Basically, it adds to the model all elements referred to by `AddedElements` of the difference model; deletes from the model all elements referred to by `DeletedElements` of the difference model; and modifies those elements of the model which are referred to by `ModifiedElements`.

In Maude, this operation can be specified in terms of three equations (described below) that correspond, respectively, to the addition, deletion and modification of elements. A fourth equation is also included to deal with the empty difference:

```
vars MODEL CONF : Configuration .
vars O O2 OLDO NEWO : Oid .
vars C NEWC OLDC : Cid .
vars ATTS OLDATTS NEWATTS : AttributeSet .

op do : Configuration DiffModel -> Configuration .
eq do(MODEL, < O : AddedElement | element : NEWO >
             < NEWO : NEWC | NEWATTS > CONF)
  = < originalId(NEWO) : NEWC | NEWATTS > do(MODEL, CONF) .
ceq do(< O : C | ATTS > MODEL,
       < O2 : DeletedElement | element : OLDO >
       < OLDO : OLDC | OLDATTS > CONF)
  = do(MODEL, CONF)
  if O = originalId(OLDO) .
ceq do(< O : C | ATTS > MODEL,
       < O2 : ModifiedElement | element : NEWO, oldElement : OLDO >
       < NEWO : NEWC | NEWATTS > < OLDO : OLDC | OLDATTS > CONF)
  = < originalId(NEWO) : NEWC |
       (excludingAll(ATTS,OLDATTS), NEWATTS)) > do(MODEL, CONF)
  if O = originalId(OLDO) .
eq do(MODEL, none) = MODEL .
```

Operation `originalId` recovers the original identifier of the object that was modified, i.e., reverts the changes done by operations `newId` or `oldId` in the model difference. Operation `excludingAll` (used in the third equation), deletes from a Maude attribute set `ATTS` all attributes that have the same name of a given attribute in the `OLDATTS` set. Since `OLDATTS` just contains the attributes that have to be deleted or that were modified, what we are doing here is removing from `ATTS` these elements just to add the `NEWATTS` set later. The `NEWATTS` set

contains all the attributes to be added, and the new values of the modified attributes, so that the elements are properly built.

Note that a matching between both models (M_s and M_d) is not needed, because operation `do` is supposed to be applied to the original model of the difference, and original object identifiers can be recovered from the difference model.

The resulting model M_m will usually conform to the same metamodel MM of M_s, although since model difference can be applied to models conforming to different metamodels, in general we can just affirm that the resulting model M_m will conform to a metamodel which is a *subtype* [1] of the metamodel MM of the original subtrahend M_m.

5.2 The "undo" Operation

Given a model M_m conforming to a metamodel MM and a difference model M_d (conforming to the *Difference Metamodel*), the result of applying operation `undo` to them is another model M_m so that: `undo(`M_m`,`M_d`) =` M_s and `modelDiff(`M_m`,`M_s`) =` M_d. As well as in operation `do`, the resulting model M_s will usually conform to the same metamodel MM of M_m (if this was true when the difference was done).

This operation reverts all the changes specified in a difference model. Basically, it adds to the model all elements referred to by `DeletedElements` of the difference model; deletes from the model all elements referred to by `AddedElements` of the difference model; and modifies those elements of the model that are referred to by `ModifiedElements` (but in the opposite way of operation `do`). `Undo` equations are not shown here because they are analogous to the `do` equations.

Operation `undo` can be considered as the inverse operation of `do`. Thus, `undo(do(`M_s`,` M_d`),` M_d`) =` M_s, and `do(undo(`M_m`,` M_d`),` M_d`) =` M_m. This is always true because of the definition of both operations: `do(`M_s`,` M_d`) =` M_m, and `undo(`M_m`,` M_d`) =` M_s.

5.3 Sequential Composition of Differences

Another important operation provides the sequential composition of differences. In general, each *difference model* represents the changes in a model from one version to the next, i.e., a *delta* (Δ). The `diffComp` operation specifies the composition of deltas, so that individual deltas can be combined into a single one.

This operation is very useful, for instance, to "optimize" the process of applying successive modifications to the same model, which might introduce complementary changes. For example, if one element is added in one delta and then deleted in another, the composed delta does not need to store both changes. In this way, this operation not only composes delta but also eliminates unnecessary changes and provides more compact model differences, hence improving efficiency.

The following Maude equations are a fragment of the `diffComp` operation specification. The first equation corresponds to the composition of an addition

and a deletion of the same element. In this case, as mentioned before, there will be no evidence of the element in the resulting difference model. The second equation corresponds to the composition of an addition and a modification of the same element. Thus, both an `AddedElement` that refers to the element with its attributes properly modified, and the element itself, will be included in the resulting difference model.

```
ceq diffComp(< O1 : AddedElement | element : NEWO >
             < NEWO : NEWC | NEWATTS > CONF1,
             < O2 : DeletedElement | element : OLDO >
             < OLDO : OLDC | OLDATTS > CONF2)
   = diffComp(CONF1, CONF2)
   if originalId(OLDO) == originalId(NEWO) .

ceq diffComp(< O1 : AddedElement | element : NEWO >
             < NEWO : NEWC | NEWATTS > CONF1,
             < O2 : ModifiedElement | element : NEWO2,
               oldElement : OLDO >
             < OLDO : OLDC | OLDATTS >
             < NEWO2 : NEWC2 | NEWATTS2 > CONF2)
   = < O1 : AddedElement | element : NEWO2 >
     < NEWO2 : NEWC2 | excludingAll(NEWATTS, NEWATTS2), NEWATTS2 >
     diffComp(CONF1, CONF2)
   if originalId(NEWO) == originalId(OLDO) .
   ...
eq diffComp(CONF1, CONF2) = CONF1 CONF2 [owise] .
```

When no correspondences are found between the two difference models, i.e., there are no several `DiffElement` that refers to the same element, all the remaining elements (`DiffElements`) from both models are just copied (as specified by the last equation).

5.4 Tool Support

We have developed an Eclipse plug-in, called Maudeling [11], which is available for download [19]. This plug-in provides the implementation of all the difference operations specified here.

One of the main advantages of Maude is the possibility of using its execution environment, able to provide efficient implementations of the specifications. In fact, Maude's execution capabilities are comparable in performance and resource consumption to most commercial programming languages' environments. Thus we can efficiently execute the specifications of all the model operations described above.

Internally, ATL is used to automatically transform the Ecore models into their corresponding Maude representations, and then execute the operations in the Maude environment, so that the user does not need to deal with the Maude encoding of models and metamodels.

6 Related Work

There are several works that address the problems of comparing models and calculating their differences. Firstly, we have the works that describe how to compute the difference of models that conform to one specific metamodel, usually UML [5,9,10]. Their specificity and strong dependence on the elements and structure of the given metamodel hinders their generalization as metamodel-independent approaches that can be used with models that conform to arbitrary metamodels.

Secondly, there are several techniques that allow to solve this problem using edit scripts, which are based on representing the modifications as sequences of atomic actions specifying how the initial model is procedurally modified. These approaches are more general and powerful, and have been traditionally used for calculating and representing differences in several contexts. However, edit scripts are intrinsically not declarative, lengthy and very fine-grained, suitable for internal representations but quite ineffective to be adopted for documenting changes in MDSD environments, and difficult to compose. Furthermore, the results of their calculations are not expressed as a model conforming to a specific metamodel, and therefore can not be processed by standard modeling platforms (cf. [20]).

Other works, such as [20], [17] and [16], are closer to ours. In [20], a metamodel-independent approach to difference representation is presented, but with the particularity that the *Difference Metamodel* is not fixed, but created in each case as an extension of the operands' metamodel. This approach is agnostic of calculation method, so it does not introduce any model difference operation, and also requires a complex model transformation process to create the specific Difference Metamodel and to calculate the differences. Matching is based on name comparison; the proposal assumes that this specific attribute always exists and is called `name`. This may hinder its application in some contexts, such as for instance those in which metamodels are defined in languages different to English.

The work described in [17] introduces an algorithm for calculating and representing model differences in a metamodel-independent manner, but the result is not compact (it contains more information than required) and it is more oriented towards graphically representing and visualizing the differences.

Thirdly, EMFCompare [16] is a new interesting approach that uses complex and sophisticated algorithms to compute the structural matching between model elements. However, this proposal makes heavy use of the hierarchical tree for matching the elements, restricting the comparisons to elements in the same level. As discussed in Section 4.2, this restriction may not be effective in some cases, including those in which the changes affect the tree structure (something common in several model refactoring operations). Furthermore, difference models are not self-contained in EMFCompare, and therefore the minuend and subtrahend models are required in order to operate with the differences.

Finally, none of these approaches currently provide good support for composing the deltas and implementing further operations on them.

7 Conclusions

This paper discusses the problem of calculating differences between models conforming to arbitrary metamodels. Differences are represented as models that conform to the *Difference Metamodel*. In this way, the proposal has been devised to comply to the "everything is a model" principle.

The *Difference Metamodel*, and difference and related operations (`do`, `undo` and composition) have been specified in Maude and integrated in our Eclipse-developed environment Maudeling [2]. These operations can be applied to models not initially specified in Maude since ATL transformations have been defined from other kinds of representations, such as KM3 or Ecore.

There are several lines of research in which we are currently engaged, or that we plan to address in the near future.

Firstly, we are working on reverse transformations, i.e., transformations from Maude to Ecore model specifications, in order to make Maude completely transparent to the user and allow other tools to use the results produced by Maude.

Secondly, future work will address the problem of conflict detection and resolution in case of concurrent modification of models in distributed collaborative environments, in which parallel composition of differences can be applied to a model (or to parts of it).

Thirdly, we are working on improving the matching algorithm, with more complex heuristics and more customizable parameters, allowing users to assign weights to particular structural features depending on the specificities of their metamodels. In this sense, many of the powerful matching algorithms being developed within the EMFCompare project can also be easily adopted by our proposal, given the powerful specification possibilities of Maude. Counting on a formal supporting framework may bring along interesting benefits, such as proving the correctness of the algorithms, or reasoning about them, something at which Maude is particularly strong.

Finally, we are also working on making all our model operations available via Web services. In this way, users can simply send the appropriate SOAP messages with the (URLs of the) Ecore models to be compared, and get the resulting Ecore model with the difference. These difference models can be provided as operands of `do`, `undo` and `modelDiff` operations, also supported as Web Services. Our goal is to contribute to a Web-based distributed *Model Service Bus*, supporting a set of common services and operations on models that can be used for achieving distributed mega-programming in an effective way.

Acknowledgements. The authors would like to thank Francisco Durán for his help with the Maude system, and also to the anonymous referees for their insightful comments very constructive suggestions. This work has been supported by Spanish Research Project TIN2005-09405-C02-01.

References

1. Steel, J., Jézéquel, J.M.: Model typing for improving reuse in model-driven engineering. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 84–96. Springer, Heidelberg (2005)
2. Romero, J.R., Rivera, J.E., Durán, F., Vallecillo, A.: Formal and tool support for model driven engineering with maude. In: Meyer, B., Bézivin, J. (eds.) Proc. of TOOLS Europe 2007, Zurich, Switzerland (2007)
3. Bernstein, P.: Applying model management to classical metadata problems. In: Proc. of Innovative Database Research, pp. 209–220 (2003)
4. Cicchetti, A., Ruscio, D.D., Pierantonio, A.: A domain-specific modeling language for model differences. Technical report, Università di L' Aquila (2006)
5. Alanen, M., Porres, I.: Difference and union of models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 2–17. Springer, Heidelberg (2003)
6. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Softw. Eng. 28, 449–462 (2002)
7. Ohst, D., Welle, M., Kelter, U.: Differences between versions of UML diagrams. In: ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 227–236. ACM Press, New York (2003)
8. Brun, C., Pierantonio, A.: Model differences in the eclipse modeling framework. Upgrade, Special Issue on Model-Driven Software Development IX (2008)
9. Ohst, D., Welle, M., Kelter, U.: Difference tools for analysis and design documents. In: ICSM 2003: Proceedings of the International Conference on Software Maintenance, p. 13. IEEE Computer Society, Washington (2003)
10. Xing, Z., Stroulia, E.: Umldiff: an algorithm for object-oriented design differencing. In: ASE 2005: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pp. 54–65. ACM Press, New York (2005)
11. Rivera, J.E., Durán, F., Vallecillo, A., Romero, J.R.: Maudeling: Herramienta de gestión de modelos usando Maude. In: JISBD 2007: Actas de XII Jornadas de Ingeniería del Software y Bases de Datos (2007)
12. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.: Maude: specification and programming in rewriting logic. Theoretical Computer Science 285, 187–243 (2002)
13. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude 2.0 Manual (2003), `http://maude.cs.uiuc.edu`
14. Martí-Oliet, N., Meseguer, J.: Rewriting logic: roadmap and bibliography. Theoretical Computer Science 285, 121–154 (2002)
15. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
16. Toulmé, A.: The EMF compare utility (2007), `http://www.eclipse.org/modeling/emft/`
17. Lin, Y., Gray, J., Jouault, F.: DSMDiff: A differentiation tool for domain-specific models. European Journal of Information Systems 16, 349–361 (2007)

18. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10, 707 (1966)
19. Rivera, J.E.: Maudeling (2008), `http://atenea.lcc.uma.es/index.php/Portada/Resources/Maudeling`
20. Cicchetti, A., Ruscio, D.D., Pierantonio, A.: A metamodel independent approach to difference representation. In: Meyer, B., Bézivin, J. (eds.) Proc. of TOOLS Europe 2007, Zurich, Switzerland (2007)