Difference and Union of Models

Marcus Alanen and Ivan Porres

TUCS Turku Centre for Computer Science
Department of Computer Science,
Åbo Akademi University
Lemminkäisenkatu 14, FIN-20520 Turku, Finland
e-mail:{marcus.alanen,ivan.porres}@abo.fi

Abstract This paper discusses the difference and union of models in the context of a version control system. We show three metamodel-independent algorithms that calculate the difference between two models, merge a model with the difference of two models and calculate the union of two models. We show how to detect union conflicts and how they can be resolved either automatically or manually. We present an application of these algorithms in a version control system for MOF-based models.
Keywords: Metamodelling, Delta Calculation, Revision Control, UML

1 Introduction

Asset, version and configuration management is an important activity in any large software development project. This is still true if we use models as the main description for our software. Sooner or later, we will have not one but several related models describing the same software. These models may represent different designs of the same subsystem, different subsystems created in parallel by several designers, or a combination of both cases. If the models are large, we need special tools to compare and merge several different models into a new one that contains all the changes proposed by all the developers.

We may illustrate this problem as follows. Let us assume that the original model shown at the top of Figure 1 is edited simultaneously by two developers. One developer has focused his work on the classes A and B and decided that the subclass B is no longer necessary in the model. Simultaneously, the other developer has decided that class C should have a subclass D. The problem is to combine the work of both developers into a single model. This is the model shown at the bottom of Fig. 1.

In this article we study how to perform this operation in a generic way. The problem is not trivial. In the example, if we would just perform the union of the models, i.e., take all elements that appear in the two versions of the model, we would obtain a model that does not contain the changes proposed by the first developer. The solution is based on calculating the final model based on the differences from the original model. Figure 2 shows an example of the difference of two models, in this case the difference between the models edited by the developers and the original model. The result of the difference is not always a model, in a similar way that the difference between two natural numbers is not a natural number but a negative one. An example of this is shown in the

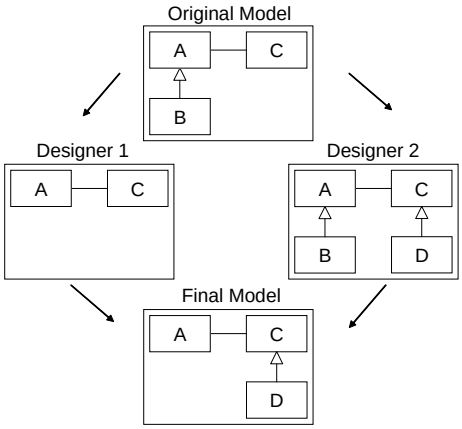


Figure1. Example of the Union of Two Versions of a Model

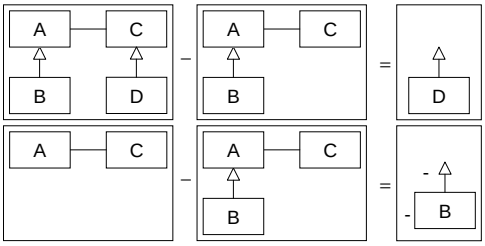


Figure2. Example of the Difference of Models

bottom part of Fig. 2. In this case, the difference of the models contains *negative* model elements, i.e., elements that should be removed from a model.

In the article we show two algorithms to calculate the difference between two models and to merge a model with a difference. Given two models M_1 and M_2 defined in UML or another modelling language based on the Meta Object Facility (MOF) [6], we define the following operations:

Difference of two models	$M_2 - M_1 = \Delta$
Merge of a model and a difference	$M_1 + \Delta = M_2$

Once we know how to operate with differences between two models, we can solve our original problem by computing the union of two versions of a model as follows:

$$M_{\text{final}} = M_{\text{original}} + (M_1 - M_{\text{original}}) + (M_2 - M_{\text{original}})$$

Figure 3 shows this operation intuitively. In practise, the implementation of this operation is complicated by the fact that the two developers may have changed the

same subset of the model. This situation can lead to conflicts and it may not be possible to apply all changes into the final model.

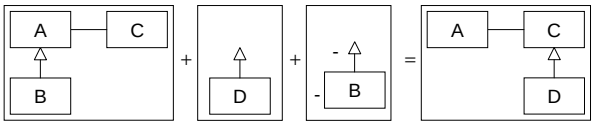


Figure3. Example of the Union Based on Differences

The presented algorithms are implemented in a generic way, i.e., the algorithms are not defined in terms of a specific modelling language but can be applied in any MOF-based modelling language. However, these algorithms crucially rely on the existence of a universally unique identifier for each model element.

These basic operations are useful in many problems that appear in model management, especially in a distributed setting. An obvious application is a version control system with optimistic locking that allows many developers to work on a model simultaneously. Also, a model repository that stores different revisions of a model may store the difference between revisions instead of complete models, saving storage space. Similarly, two computers connected by a slow link may interchange a difference between models, saving bandwidth and communication time.

The paper is structured as follows: Section 2 states the minimum requirements of the metamodel and the model elements in our modelling language. In Section 3, we explain the algorithm for calculating the difference between two models and for applying said difference to one model, producing the other. These algorithms alone can be used for saving transmission time or storage space. Section 4 explains how these algorithms are used to create the union algorithm between two models and their base model. Conflict situations in such cases are a fact, and a closer look of what kinds of conflicts can be avoided are studied. We conclude in Section 5 by stating what problems are solved, and how to build further on this work.

2 Models and Metamodels

This section describes what requirements are set for the algorithms to work. In particular, we note that these requirements are supported by the UML standard as proposed by the Object Management Group (OMG) [5].

2.1 The Metamodel Layer

A model consists of a set of linked elements. Each element in a model conforms to a type, called a metaclass, while each link between two model elements conforms to a meta-association. The meta-association is further divided into two association ends or metafeatures.

In the UML standard, the metamodel is defined using classes, that may contain attributes of a given type, generalisations between classes and associations between the classes. There is a special kind of association called composition that represent a whole/part relationship between the model elements.

Attributes and associations in the metamodel have two other properties: *multiplicity* and *order*. The multiplicity describes a constraint on the number of values that can be stored in the attribute or association end. Additionally, these elements may be ordered. An example of an ordered feature is the sequence of parameters in a method.

The complete definition of the metaclasses and meta-associations allowed in a modelling language is called the metamodel.

We assume that each basic data type used in a metamodel has a default value, called the "zero" of the metafeature. The default value of an integer is the value 0, the default value of a string is the empty string, and the default value of an enumeration is its first value. The default value of unordered and ordered associations are the empty set and empty sequence, respectively.

Although attributes are unidirectional features, it is of utmost importance to keep the bidirectionality of associations; if an element *A* has an association to *B*, then *B* must have an association back to *A*. For example, as a UML 1.4 Parameter element has a **type** association to a Classifier element, then also that Classifier element has a **typedParameter** association back to the Parameter element. Any operation on a model must preserve the association in a consistent state. The metafeature from *A* to *B* is said to have an *opposite* metafeature from *B* to *A*. An ordered metafeature can also have an unordered metafeature as its opposite metafeature; therefore, the actual contents of the ordered metafeature comprise a set, not a bag.

2.2 Unique Element Identifiers

Some parts of the algorithms are greatly simplified by requiring that each element has a universally unique identifier (UUID). In practise, several other operations on models also use such an identifier, so it is sound to take advantage of it. The textual encoding of a model using XML Metadata Interchange (XMI) [7] is a prime example of an OMG standard which uses unique identifiers. An XMI identifier is usually generated as defined by the OMG, and is of the form **namespace:uuid**. An example of an identifier in the DCE [1] namespace is "DCE:2fac1234-31f8-11b4-a222-08002b34c003". It is assumed that the UUID of an element does not change after it has been set, and the uniqueness of the generator's UUIDs is not questioned.

3 Model Difference and Merge

This section describes how to calculate the difference between two models, M_{old} and M_{new}. We represent the result of this operation as a Δ. It contains a sequence of transformations that when applied to model M_{old}, yields model M_{new}.

$$M_{\text{new}} - M_{\text{old}} = \Delta$$
$$M_{\text{old}} + \Delta = M_{\text{new}}$$

3.1 Description of a Delta

We have shown informally in the introduction that the result of the difference between two models may contain negative elements, i.e. information that should be removed from a model. We consider that it is more intuitive to represent a Δ in operational terms; not as a set of elements and negative elements but as a sequence of transformations that add or remove elements from a model.

We have identified seven elementary transformations in a model that will be used as the basis for defining a Δ. We assume that it is not possible to change the type of a model element, e.g., a UML Class cannot become a Package, and an element cannot change its UUID. The operations in a Δ are:

- Element creation and deletion.
  - new(e,t) : Create a new element of type t with the UUID of e. By default, a new element has all its features set to their default values.
  - del(e,t) : Delete an element of type t with the UUID of e. An element may only be deleted if all its features are set to their default values.
- Modification of a feature.
  Modification of a feature of type f of an element e with UUID u. Where necessary, there is another element e_t with UUID u_t. Depending on the type of the feature, this might mean one of the following modifications:
  - set(e,f,v_o,v_n) : Set the value of $e.f$ from v_o to v_n, for an attribute of primitive type.
  - insert(e,f,e_t) : Add a link from $e.f$ to e_t, for an unordered feature.
  - remove(e,f,e_t) : Remove a link from $e.f$ to e_t, for an unordered feature.
  - insertAt(e,f,e_t,i) : Add a link from $e.f$ to e_t, at index i, for an ordered feature.
  - removeAt(e,f,e_t,i) : Remove a link from $e.f$ to e_t, which is at index i, for an ordered feature.

  It should be noted that none of the operations try to maintain bidirectionality of associations. It is maintained at a higher level in the actual difference algorithm. Also, for transferring operations over the network, the UUID of an element must be used instead of the actual element.

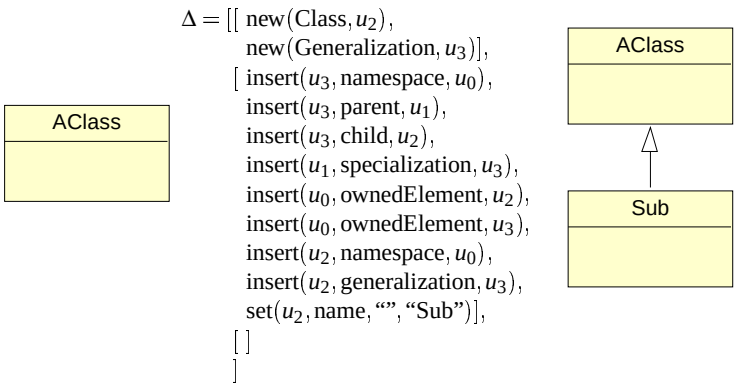
These operations have three properties. First, the positive operations (new, set, insert and insertAt) are complete in the sense that they can be used to represent any model. Also, each operation has a dual operation with the opposite effect. The map between operations and their dual operations is given in Table 1. This is needed to calculate the inverse of a Δ. Finally, the new and del operations do not contain references to other elements, which will simplify the construction of algorithms that work with a Δ.

For the remainder of this paper, we use the notation $[a,b,c,\ldots]$ for sequences, and $\{a,b,c,\ldots\}$ for sets. The Δ is partitioned into three separate sequences, $\Delta = [C,F,D]$ such that all new(e,t) operations are in C, all del(e,t) operations are in D, and the rest are in F.

An example of a difference between two models is given in Figure 4, in which the old model is on the left and the new model is on the right. In the Δ, two new elements u_2 and u_3 are created. They are connected to the root Model element u_0 (not shown) via their **namespace** association, and the Model connects them to its **ownedElement**

Operation O	Dual operation $\tilde{O}$
new(e,t)	del(e,t)
del(e,t)	new(e,t)
set(e,f,v_o,v_n)	set(e,f,v_n,v_o)
insert(e,f,e_t)	remove(e,f,e_t)
remove(e,f,e_t)	insert(e,f,e_t)
insertAt(e,f,e_t,i)	removeAt(e,f,e_t,i)
removeAt(e,f,e_t,i)	insertAt(e,f,e_t,i)

Table1. The Map Between Operations and Dual Operations.



$$
\begin{aligned}
\Delta = [[\ & \text{new}(\text{Class}, u_2),\\
& \text{new}(\text{Generalization}, u_3)],\\
[\ & \text{insert}(u_3, \text{namespace}, u_0),\\
& \text{insert}(u_3, \text{parent}, u_1),\\
& \text{insert}(u_3, \text{child}, u_2),\\
& \text{insert}(u_1, \text{specialization}, u_3),\\
& \text{insert}(u_0, \text{ownedElement}, u_2),\\
& \text{insert}(u_0, \text{ownedElement}, u_3),\\
& \text{insert}(u_2, \text{namespace}, u_0),\\
& \text{insert}(u_2, \text{generalization}, u_3),\\
& \text{set}(u_2, \text{name}, \text{“”}, \text{“Sub”})],\\
[\ &]\\
]
\end{aligned}
$$

Figure4. Difference Between Two Simple Models.

composition, due to bidirectionality constraints. The new class u_2 is connected to the old class u_1 via the Generalization element, using its **specialization** and **generalization** features. Since all features start with their respective “zero” value, the **name** feature of the new class is also set by the difference.

XMI has facilities for representing arbitrary differences between two models, using an XML element called **XMI.difference**. The positive operations new, insert and insertAt can be described in a XMI document using the **XMI.add** element, while the negative operations del, remove and removeAt can be specified using the **XMI.delete** element. The set operation can be represented using the **XMI.replace** element. XMI also specifies that differences must be applied in the order defined, which is also a requirement of the algorithms in this paper.

3.2 Difference Algorithm

Once we know how to represent the difference between two models, we can describe an algorithm to calculate it. The proposed difference algorithm has four steps, as discussed in [2]. The objective is first to create an unambiguous mapping between the elements in

M_{old} and in M_{new} and then calculate an exact sequence of operations that can transform each element in M_{old} to the corresponding one in M_{new}.

Map This phase creates a mapping between elements in M_{old} and M_{new}. In our case, the UUIDs of the elements serve as the map. From this, we create $E_{\text{old}}(u) = e$ such that u is the UUID of $e \in M_{\text{old}}$ and $E_{\text{new}}(u) = e$ such that u is the UUID of $e \in M_{\text{new}}$. It is possible to create such a mapping without relying on the UUIDs, but it would be a lot harder and resource-intensive. However, it could yield a smaller Δ.

Create The Δ should contain an operation to create each element in M_{new} that does not exist in M_{old}. Given our mapping between elements in M_{old} and M_{new}, we define the sequence C of elements that need to be created as follows:

$$C := [\text{new}(e,t) \mid \forall e,t : e = E_{\text{new}}(\text{dom}(E_{\text{new}}) \setminus \text{dom}(E_{\text{old}})) \wedge t = \text{typeof}(e)]$$

Delete Similarly, the Δ should contain an operation to delete each element in M_{old} that does not exist in M_{new}. Again, the sequence D of elements that need to be deleted is easy to define:

$$D := [\text{del}(e,t) \mid \forall e,t : e = E_{\text{old}}(\text{dom}(E_{\text{old}}) \setminus \text{dom}(E_{\text{new}})) \wedge t = \text{typeof}(e)]$$

After defining the sequences C and D, it is necessary to update the map between M_{old} and M_{new} to be bijective. This is accomplished by updating E_{old} and E_{new} to have the same domain, by adding new elements into E_{old} which are in E_{new} but not in E_{old}, and vice versa. The created elements have default values for all their features. Now there exists a set P of tuples (e_o, e_n) such that for each element $e_o \in \text{range}(E_{\text{old}})$ there exists an element $e_n \in \text{range}(E_{\text{new}})$ with the same UUID as e_o.

Change In this phase we match the features for each pair of elements $(e_o, e_n) \in P$, both with the UUID u. This is done by creating a sequence of operations for each kind of feature f, which are all added to a sequence F. Applying the operations modifies $e_o.f$ into $e_n.f$.

Elements e_o and e_n have the same type, and thus the same set of features. Then, for each metafeature f in the type of e_o, create operations based on the following:

- For an attribute feature of primitive type, if the values of $e_o.f$ and $e_n.f$ differ, create an operation $\text{set}(e, f, e_o.f, e_n.f)$.
- For an unordered feature, create the following operations:
  $[\text{insert}(e, f, e_t) \mid \forall e_t : e_t \in e_n.f \setminus e_o.f,\ \text{remove}(e, f, e_t) \mid \forall e_t : e_t \in e_o.f \setminus e_n.f]$.
- For an ordered feature, the element order must be preserved. The smallest sequence of changes that transform one sequence into another is equivalent with the *Longest Common Subsequence* problem, to which there exists efficient solutions, e.g., by Myers [4]. The result is a sequence of insertAt and removeAt operations which, when applied to $e_o.f$, transforms it into $e_n.f$ at minimal size cost. The sequence has length $N + M - 2L$, where N and M are the lengths of the features, and L is the length of the longest common subsequence of N and M. The operations in the sequence should be added to F.

A complete Δ between two models is then specified by a sequence of all operations as created by the above algorithm. This is a sequence of element creations, feature modifications and element deletions, in that order, and so $\Delta = [C, F, D]$. Such a Δ should guarantee two properties:

- For each delete operation in D, there are a set of operations in F that reset the features of the elements to be deleted to their default value.
- The complete set of operations in F maintains the associations in a consistent state. The individual operations in F only update one association end. However, we should ensure that for each operation in F that updates an association end, there is a corresponding operation that updates the opposite end.

The algorithm proposed in this section satisfies these properties.

3.3 Merge Algorithm

To merge a difference to a model is to apply the transformations contained in a Δ to a model. Given a model M_{old} and a difference Δ, we denote the merge operation as $M_{\text{old}} + \Delta$ or $\Delta(M_{\text{old}})$.

Given a $\Delta = [C, F, D]$, the merge algorithm has three steps that should be performed in this order:

1. $\forall \text{new}(e,t) \in C$: Create an element of type t with the UUID of e.
2. $\forall o \in F$: Make the feature change o.
3. $\forall \text{del}(e,t) \in D$: Delete the element of type t with the UUID of e.

The actual implementation of these transformations depends on the action language used to transform the models. A requirement of a metamodel is that we have a reflection interface for determining and querying the metafeatures of all metaclasses, and a facility for modifying the features of model elements.

3.4 Inverse of a Delta

Given two models M_{old} and M_{new} and a difference $\Delta = [C, F, D]$ such that $M_{\text{old}} + \Delta = M_{\text{new}}$, we can calculate the inverse of a difference $\tilde{\Delta}$ such that $M_{\text{new}} + \tilde{\Delta} = M_{\text{old}}$.

Calculating the inverse of a difference is a simple process that only requires to reverse the sequences in Δ and replace each operation with its dual.

$$
\begin{aligned}
\tilde{\Delta} &= [\tilde{C}, \tilde{F}, \tilde{D}] \\
\tilde{C} &= [\tilde{c}_0 \ldots \tilde{c}_{\#D-1}, \tilde{c}_i = \text{dual}(d_{\#D-i-1})] \\
\tilde{F} &= [\tilde{f}_0 \ldots \tilde{f}_{\#F-1}, \tilde{f}_i = \text{dual}(f_{\#F-i-1})] \\
\tilde{D} &= [\tilde{d}_0 \ldots \tilde{d}_{\#C-1}, \tilde{d}_i = \text{dual}(c_{\#C-i-1})]
\end{aligned}
$$

Once we have calculated the inverse of a Δ, it can be applied as described in the merge algorithm.

4 Union of Models Based on Model Differences

An interesting problem emerges when two differences, Δ_1 and Δ_2, should be applied onto the same model. This occurs frequently in a distributed development environment, or with a repository where elements under development cannot be locked before modification.

The objective is to apply Δ_1 first, then apply Δ_2. It should be noted that the result ought to be the same no matter how the actual differences are applied,

$$M_{\cup} = \Delta_1(\Delta_2(M_{\text{base}})) = \Delta_2(\Delta_1(M_{\text{base}}))$$

However, since the differences are calculated relative to M_{base}, applying one Δ first would create a model which is different from the base model, and the other Δ could not be applied as such. To see why this is true, consider an ordered feature with elements $[B,C]$ and differences $\text{insertAt}(A,0)$ and $\text{insertAt}(D,2)$. Applying the differences would create either the correct $[A,B,C,D]$ or the incorrect $[A,B,D,C]$, depending on the order in which the differences were applied.

Another example is a set $\{A,B\}$ and differences $\text{insert}(C)$ and $\text{insert}(C)$. The second operation is spurious, since the first difference already accomplishes the task: adding C to the set. Removing the other operation shortens Δ.

Thus, we need a reliable method to modify a Δ according to another Δ, and a shorter Δ is naturally preferred. The modification is necessary to avoid errors. We define the *difference minimisation operator* $\otimes$:

$$\Delta_2' = \otimes(\Delta_2, \Delta_1), \quad \Delta_1' = \otimes(\Delta_1, \Delta_2)$$

Now the equation becomes:

$$M_{\cup} = \Delta_1'(\Delta_2(M_{\text{base}})) = \Delta_2'(\Delta_1(M_{\text{base}}))$$

This principle is also illustrated in Figure 5. Without loss of generality, this paper discusses only the calculation of $\Delta_2' = \otimes([C_2,F_2,D_2],[C_1,F_1,D_1])$.

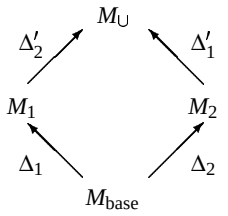


Figure5. The principle of calculating the union of two models, given their base model. Either difference is modified according to the other one, and then applied.

4.1 Difference Minimisation

The calculation of $[C_2', F_2', D_2'] = \otimes([C_2, F_2, D_2], [C_1, F_1, D_1])$ has several different cases: element creation and deletion, changes to the attributes, unordered features and ordered features. Given our method of modifying each feature of each element with a small sequence of operations, the calculation of F_2' also happens one feature at a time. The difference minimisation methods of the various kinds of operations, and which conflicts can occur, are presented in the following subsections. A common resolution is to ignore an operation $o \in F_2$ instead of adding it to F_2'. This also implies that for bidirectional features, the operation for the opposite feature must have the same resolution as o, properly ignored or added to F_2'.

Element Creation and Deletion The set of elements created in Δ_2 are unaffected by any operations in Δ_1. Due to the UUID generator's uniqueness, the operations in Δ_1 cannot refer to the new elements in Δ_2. Therefore, all new(e,t) operations in Δ_2 are valid.

Element deletion can be a source of conflicts. For each del(e,t) operation in Δ_2, if Δ_1 also has the same operation, we can remove it from Δ_2, since Δ_1 will already delete the element. The worst case occurs when one Δ modifies e without deleting it, and the other Δ has an operation for deleting it. One solution would be to ignore the modifications and delete the element. However, it is questionable if this is the correct behaviour and has the intended effect every time, so manual resolution might be the only viable choice.

$$C_2' = C_2$$
$$D_2' = [o \mid o \in D_2 \wedge o \notin D_1] \quad \text{(in general)}$$

Setting Attributes Difference minimisation calculations for attributes is straightforward. There is a conflict if both differences try to change the same attribute feature of primitive type with operations set$(e, f, v_o, v_{n_1}) \in F_1$ and set$(e, f, v_o, v_{n_2}) \in F_2$ and $v_{n_1} \neq v_{n_2}$. Then, either the operation is not added to F_2', or it takes precedence as set(e, f, v_{n_1}, v_{n_2}), as requested by the user. If $v_{n_1} = v_{n_2}$, the operation does not have to be added to F_2', since the operation in F_1 already does it.

For operations set$(e, f, v_o, v_{n_1}) \in F_2$ where set$(e, f, v_o, v_{n_1}) \notin F_1$, the operation can be added as such to F_2'. Since attribute features are unidirectional, there is no opposite feature that must be updated as well.

$$\otimes(F_1, F_2) = [o \mid o(e, f, v_o, v_{n_2}) \in F_2 \wedge o(e, f, v_o, v_{n_1}) \notin F_1 \wedge o \text{ is set}]$$

Unordered Features Each remove(e, f, e_t) operation in F_2 can be added to F_2' if it does not exist in F_1. Each insert(e, f, e_t) operation in F_2 can be added to F_2' if it does not exist in F_1. For bidirectional features, it is important to only add these operations to F_2' if a similar operation can be added to F_2' for the opposite feature, which can be unordered or ordered.

If there are no multiplicity constraints on the feature, no conflicts are possible. In particular, many features have a multiplicity constraint of "at most one element". If both differences add elements to such a feature, manual resolution must choose which one of the operations should prevail, but otherwise the following suffices:

$$\otimes(F_1, F_2) = [o \mid o \in F_2 \wedge o \notin F_1 \wedge (o \text{ is remove} \vee o \text{ is insert})]$$

Ordered Features The idea of difference minimisation for ordered features is to interleave the insertAt and removeAt operations in Δ_2 with the ones in Δ_1. This is accomplished one feature at a time, so we work with insertAt and removeAt operations $[f_0, \ldots, f_{m-1}]$, $f_i \in F_2$, and $[g_0, \ldots, g_{n-1}], g_i \in F_1$. The original sequence of elements is denoted F_{orig}. It is not straightforward which operations in F_2 should be added into F_2', and as we have seen in previous examples in the introduction in section 4, the indices of the individual f_i operations might need to be modified.

Currently, this is accomplished by our algorithm in Figure 6. It takes as input parameters two sequences of operations F_1 and F_2 which contain operations $\text{insertAt}(e, f, e_t, i)$ and $\text{removeAt}(e, f, e_t, i)$ for a fixed element e and feature f. The operations in each sequence come in index order, beginning from the smallest index, and with insertions occurring before deletions. Because conflicts are inevitable, the output is a set of sequences, from which the user must choose one solution, whose operations will be added to F_2'. In the algorithm, $[]$ corresponds to sequence indexing, $\leftarrow$ to assignment, $=$ to comparison, and $|s|$ to the length of sequence s.

$$\otimes(F_1, F_2) = \text{difference\_minimisation\_for\_ordered\_feature}(\mathrm{F}_1, \mathrm{F}_2)$$

The algorithm works by scanning through F_2 while keeping track of the changes in F_1. Most notably, it removes the operations in F_2 which were already performed in F_1, and interleaves the other operations by modifying the indices, to form F_2'.

Two integers s_1 and s_2 index into each sequence's operations. Additionally, we keep offset values a_1 and a_2 which track how many insertions and deletions have occurred in the other sequence, from indices 0 (inclusive) to s_i (exclusive). An offset value tells how much an index has to be adjusted to reveal its real position, its *relative index*, in the unified sequence. As an example, an insertion with a low index in F_1 will early increase a_2 by one, and therefore all operations in F_2 will afterwards be adjusted one index forward, to keep F_2 in synchronisation with the operations in F_1.

At every iteration, the current operations $o_1 = F_1[s_1]$ and $o_2 = F_2[s_2]$ are retrieved, and their relative indices are compared (lines 4, 8 and 13). If o_1 occurs before o_2, the offset a_2 is modified according to the command in o_1, and the next operation in F_1 is taken (lines 5–7). If o_2 occurs before o_1, the offset a_1 is modified according to the command in o_2, and the next operation in F_2 is taken (lines 9–12).

When the current operations occur at the same index, a closer look at the actual operations is taken. If both operations remove an element, it must be the same element they are removing, so the operation can be removed from F_2 (lines 14–17). If o_1 removes and o_2 inserts an element, the insertion takes precedence and s_2 advances to the next operation, while the offset a_1 is increased (lines 18–19). Similarly, if o_1 inserts and o_2 removes an element, s_1 advances to the next element and a_2 is increased (lines 35–36).

```
1 function worker(F1, F2, s1, s2, a1, a2):
2     while s1 < |F1| and s2 < |F2|:
3         o1, o2 ← F1[s1], F2[s2]
4         if o1.index + a1 < o2.index + a2:
5             if o1.command = removeAt: a2 ← a2 - 1
6             if o1.command = insertAt: a2 ← a2 + 1
7             s1 ← s1 + 1
8         else if o1.index + a1 > o2.index + a2:
9             if o2.command = removeAt: a1 ← a1 - 1
10            if o2.command = insertAt: a1 ← a1 + 1
11            o2.index ← o2.index + a2
12            s2 ← s2 + 1
13        else:
14            if o1.command = removeAt:
15                if o2.command = removeAt:
16                    F2.remove(s2)
17                    s1 ← s1 + 1
18                else if o2.command = insertAt:
19                    a1, o2.index, s2 ← a1 + 1, o2.index + a2, s2 + 1
20            else:
21                if o2.command = insertAt:
22                    if o1.value = o2.value:
23                        F2.remove(s2)
24                        s1 ← s1 + 1
25                    else:
26                        results_a ← worker(F1, F2, s1+1, s2, a1, a2+1)
27                        F2[s2].index ← F2[s2].index + a2
28                        results_b ← worker(F1, F2, s1, s2+1, a1+1, a2)
29                        if |results_a[0] < |results_b[0]|:
30                            return results_a
31                        else if |results_a[0]| > |results_b[0]|:
32                            return results_b
33                        else:
34                            return results_a ∪ results_b
35                else:
36                    a2, s1 ← a2 + 1, s1 + 1
37    while s2 < |F2|:
38        F2[s2].index ← F2[s2].index + a2
39        s2 ← s2 + 1
40    return [ F2 ]
41
42 function difference_minimisation_for_ordered_feature(F1, F2):
43    return worker(F1, F2, 0, 0, 0, 0)
```

Figure6. Algorithm for modifying F_2 of an ordered feature, given that F_1 has already been applied.

If both operations insert an element, and it happens to be the same element, we can take a small shortcut by removing the relevant operation from Δ_2 (lines 22–24). Otherwise, the only real conflict occurs when both o_1 and o_2 have insertion operations at the same index: either o_1 or o_2 should take precedence, but we do not know which would create a shorter modified Δ_2. Thus, we recurse with both variants (lines 26–28) and choose the difference which creates a shorter, modified Δ_2 (lines 29–34). Finally, we make sure that all operations in Δ_2 have their indices modified by a_2 (lines 37–39).

The most common conflict occurs when both differences insert different elements at the same index. Of importance is also to remember that the resulting sequence $F_2'(F_1(F_{\text{orig}}))$ cannot contain the same element twice. The algorithm does not check for such possibilities when interleaving the operations, so the resulting sequence must be inspected for such occurrences, and the user must choose which elements to remove, before modifying the feature $e.f$.

Figure 7 shows an example of the algorithm for interleaving ordered features.

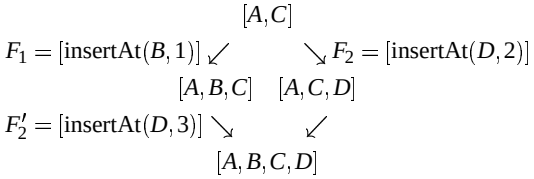


Figure7. Example of a merge of an ordered sequence. Note how the insertion of B at index 1 pushes the insertion of D from index 2 to index 3.

4.2 A Version Control System

Metamodels have, in addition to the constraints expressible by MOF, a set of well-formed rules (WFR) which determine if a model is a valid instance of the metamodel. On a metamodel-independent level, the WFRs of a metamodel cannot be kept. Therefore, even a successful, conflict-free union can still be invalid in the rules of the metamodel. In these cases manual resolution may seem like the only choice, but metamodel-specific resolution may also automatically resolve some of the problems by analysing the resulting (non-well-formed) model, and modifying it to a well-formed state.

One example of a metamodel-specific resolution mechanism presents itself with merging diagram information. The diagram elements themselves do not have any semantic meaning, so the features of the diagram elements are not nearly as correctness-critical as the underlying model. For example, conflicting diagram element coordinates on the diagram canvas can more or less be completely ignored by removing or modifying the relevant operations from Δ_2. Clearly, there is a strong need for metamodel-specific resolvers.

The schema in Figure 8 summarises a version control system for models. The difference under modification, Δ_2, passes through several filters which modify it to better

fit $\Delta_1(M_{\text{base}})$. Obviously, all possible mechanic resolution mechanisms should be tried before manual resolution is used. The algorithms described in this paper work as the first, metamodel-independent filter.

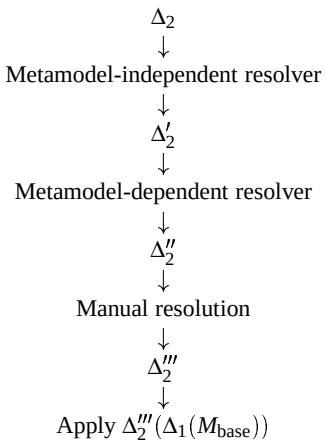


Figure8. A complete merging system with three distinct resolution steps.

The algorithm in this section can be further extended. Given a base model M_{base} and n differences $\Delta_1, \Delta_2, \ldots, \Delta_n$, we notice that the amount of differences can be reduced by taking the union $M = \Delta_2'(\Delta_1(M_{\text{base}}))$, and calculating a difference $\Delta_{1'} = M - M_{\text{base}}$. Now we have the same base model M_{base} and $n-1$ differences $\Delta_{1'}, \Delta_3, \Delta_4, \ldots, \Delta_n$. Iterating through this algorithm we have the final model $M_{\cup}$. This is important in a repository of a version control system for models, where several developers base their work on some common base model, and later commit it back to the repository, merging their changes with the work of others.

It remains to be investigated if the above mechanism is feasible, or if a merging algorithm is required which would consider more differences in parallel [3] [8].

5 Conclusions and Related Work

This work is not specific to UML but to MOF. It has presented several metamodel-independent algorithms regarding difference calculation between models. We have described a difference calculation algorithm between two models and a merging algorithm for applying the difference to the original model to produce the target model. Additionally, we have shown how to make the dual operations. These algorithms work using differences represented as a sequence of operations. The set of operations is minimal, complete and each operation has a dual.

Furthermore, the difference calculation is extended to form a union algorithm, where two separate modifications are made to a base model, and the union algorithm combines

both differences into one model by properly interleaving, where possible, the operations in the latter difference with the former difference. At all parts of the difference calculations, distinguishing unordered metafeatures from ordered ones is important, since difference calculation becomes much easier with unordered metafeatures. Ignoring the order criteria leads to very fast change detection for hierarchical information [9], and this has also been researched by [10].

Additionally, we have shown how to use these algorithms to create a version control system. However, these basic algorithms should be extended to support metamodel-specific resolution mechanisms.

There are several cases where merge conflicts are a fact and manual resolution is required. Modifying the same attribute or the same ordered feature easily creates such situations. For association features, the opposite feature must also be kept in synchronisation. The extreme case of deleting an element even though another difference merely modifies it slightly leads to a complex question; which difference should be prioritised? Further work in this area is clearly required as automatic conflict resolution can be considered important in a modelling framework.

Acknowledgements

We would like to thank the reviewers for their suggestions to this paper. Also, we would like to acknowledge and thank Johan Lilius for his contribution to the work that has lead us to this article, and Ion Petre for helpful advice.

References

1. CAE Specification. DCE 1.1: Remote Procedure Call, 1997. Available at `http://www.opengroup.org/onlinepubs/9629399/toc.htm`.
2. Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change Detection in Hierarchically Structured Information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, 1996.
3. Tom Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.
4. Eugene W. Myers. An O(ND) Difference Algorithm and Its Variations. *Algorithmica*, 1(2):251–266, 1986.
5. Object Management Group. `http://www.omg.org/`.
6. OMG. Meta Object Facility, version 1.4, April 2002. Document formal/2002-04-03, available at `http://www.omg.org/`.
7. OMG. XML Metadata Interchange, version 1.2, January 2002. Available at `http://www.omg.org/`.
8. Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. Parallel Changes in Large Scale Software Development: An Observational Case Study. In *Proceedings of the International Software Engineering Conference*, April 1998.
9. Yuan Wang, David J. DeWitt, and Jin-Yi Cai. X-Diff: An Effective Change Detection Algorithm for XML Documents. 2001. Submitted for publication. Available at `http://citeseer.nj.nec.com/449452.html`.
10. Albert Zündorf, Jörg P. Wadsack, and Ingo Rockel. Merging Graph-Like Object Structures. In *Proceedings of the Tenth International Workshop on Software Configuration Management*, 2001.