Supporting Collaborative Development in an Open MDA Environment[1]

Prawee Sriplakich, Xavier Blanc, Marie-Pierre Gervais
Laboratoire d'Informatique de Paris 6
8, rue du Capitaine Scott, 75015, Paris, France
Tel: +33 1 44 27 88 61, Fax : +33 1 44 27 87 71
{Prawee.Sriplakich, Xavier.Blanc, Marie-Pierre.Gervais}@lip6.fr

Abstract

The MDA approach aims to ease software maintenance faced with platform and business evolution. In this approach, main development artifacts, i.e. models, are defined with the Meta Object Facility (MOF) standard. To support collaborative development in MDA, we propose a mechanism for merging concurrent changes to MOF models. Our approach has the following novel functionality. First, as MOF models can have ordered relations, our mechanism can identify the order changes in MOF models, detect the conflicts caused by concurrent order changes, and integrate those changes. Second, as MOF models must respect multiplicity constraints, our mechanism detects the concurrent modifications that result in multiplicity violations. Therefore, it ensures the consistency of the merge result. Third, we offer a framework for building conflict resolution programs dedicated to developers' particular requirements. This framework offers a flexible and automated way for resolving conflicts.

This work is a part of ModelBus, an open environment for CASE tool interoperability. Its contribution is to enable models to be concurrently modified by several developers and with different tools. ModelBus implementation is available as the Eclipse open source project, Model Driven Development integration (MDDi).

Keywords: Merging, Software Configuration Management, Collaboration, CASE tool, MDA, MOF, Metamodel, Model

1. Introduction

The development and maintenance of complex software requires the collaboration of several developers. A well-known approach to support this collaboration is *copy-modify-merge* [20], which enables several developers to concurrently edit software documents (e.g. code, models and documentation). In this approach, a developer copies a document from the repository to his (her) environment for editing it locally. Once he finishes editing, he merges this local copy with the original copy located at the repository. Merging enables his modification to be integrated with other concurrent modifications that have been made by other developers.

The copy-modify-merge approach requires two main steps. The first step, *delta calculation*, enables the extraction of the modification (or *delta*) made locally to a document copy. The second step, *delta integration*, enables the integration of the delta with the repository-side document. The delta integration must be aware of the conflicts between this delta and other deltas that may have been integrated by other developers. These conflicts need to be solved either manually or automatically by software merge tools.

In this work, we aim to apply the copy-modify-merge approach in the context of Model Driven Architecture (MDA) [14]. In MDA, models are main development artifacts that are used for describing various aspects of software and for automating code generation. Not only are they used for producing new software but also for maintaining existing one, faced with platform and business evolution. The structures of models are defined by the Meta Object Facility (MOF) standard [15]. MOF is similar to the graph formalism: a

[1] The work presented in this paper is supported by the project MODELWARE, co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2002-2006).

model is a graph consisting of nodes and links. Moreover, MOF offers two features that enhance the expressivity of model structures:

1) A node can have an *ordered association end* that refers to a sequence of other nodes. The order of this sequence has meanings in the model's semantic.

2) Multiplicity is used for restricting the number of links between nodes.

To apply the copy-modify-merge approach to MDA, we adapt delta calculation and delta integration mechanisms to those MOF features. Compared to existing works, our mechanisms offer the following novel functionality:

- *Supporting ordered association ends*. Delta calculation mechanisms for software diagrams (i.e. models) have already been proposed by [12] [13]. However, their mechanisms do not propose the manipulation of ordered association ends in models. On the other hand, our mechanism can extract changes to link sequences, e.g. inserting links to a specific position or reordering links.

Moreover, if link sequences' orders were ignored, delta integration could produce arbitrary link orders in the result model. Our delta integration mechanism manages properly order changes according to the developer's intent. It enables the same link sequence to be concurrently edited (if those modifications affect different elements of the sequence). Otherwise, it also detects the conflicts caused when the concurrent modifications affect the same elements in the sequence.

- *Ensuring the result's consistency*. Inconsistency has been recognized as a problem of concurrent modifications [10]. In this work, our delta integration mechanism solves the inconsistency problem for MOF models, in particular, as regards multiplicity constraints.

- *Flexible and automated conflict resolution*. One way to solve conflicts is to drop/alter a subset of conflicting modifications [8]. Performing this task manually can be time-consuming. We offer a framework to build programs for automating conflict resolution according to the developer-specific rules. This approach reduces manual work on conflict resolution while accommodating developers' different requirements.

This work is a part of ModelBus [2], an open environment for CASE tool interoperability. Its contribution is to enable several developers to use different CASE tools to realize a collaborative development (i.e. they can concurrently integrate their contributions to the shared models). ModelBus implementation is available as the Eclipse open source project, Model Driven Development integration (MDDi, http://www.eclipse.org/mddi).

This paper is organized as follows. Section 2 presents an illustrative example of MOF model merging to which we refer throughout the paper. The difficulties dedicated to MOF model merging that we aim to solve are presented in section 3. In section 4, we describe our approach and its application in ModelBus for tool interoperability. Section 5 discusses related works, before conclusion.

2. A MOF model merging

2.1. MOF metamodels

MOF is a language for defining structures of any kinds of models. Those structures are called *metamodels*. A metamodel contains a set of metaclasses and associations. A *metaclass* defines the structure of a node type and the information that can be contained in each node of this type. It contains a set of properties that define the primitive data (e.g. string, integer, boolean) contained in a node.

An *association* defines the links between nodes. It has two ends, each of which corresponds to a metaclass. An association relating metaclasses `C1`, `C2` means that a node of type `C1` contains a set of references to nodes of type `C2` (and vice versa). MOF semantic requires that the values of two opposite association ends be kept consistent. E.g. if `C1` refers to `C2` then `C2` must refer back to `C1` via the opposite association end.

MOF allows one of the two opposite association ends to be specified as ordered, meaning that the reference set for this end is ordered (i.e. this set becomes a sequence). An association can also have a multiplicity constraint specifying the admissible size of the reference set, in terms of upper and lower bounds.

2.2. An illustrative example

Throughout this paper, we will illustrate the problem and our solution through a merging example. The models to be merged are Java Platform Specific Models (Java PSM), which can be used to generate Java code, according to the MDA principles.

A simplified version of the Java metamodel, which defines the Java model structure, is showed fig. 1. This metamodel captures Java concepts (e.g. Class, Field, Method, Statement) and their relations (e.g. a class contains methods and fields; a field is typed by a class; a method contains a sequence of statements).

Please note that our approach is not specific to this metamodel. This metamodel has been chosen for its simplicity, yet, it illustrates the MOF features that we highlight: 1) the multiplicity, which ensures the consistency of models (e.g., a field must be contained by exactly one class), and 2) the expression of orders in models (e.g. order of statements in a method).

We also show three Java models corresponding to the metamodel. They represent a base version and its two variants, which are result of concurrent changes by two developers. We represent Java models with the UML class diagram notation. A Java field is presented either by an attribute inside a UML class (e.g. field `name`) or by a one-way association (e.g. field `p1`). A note attached to a Java class represents its methods' contents (statements).

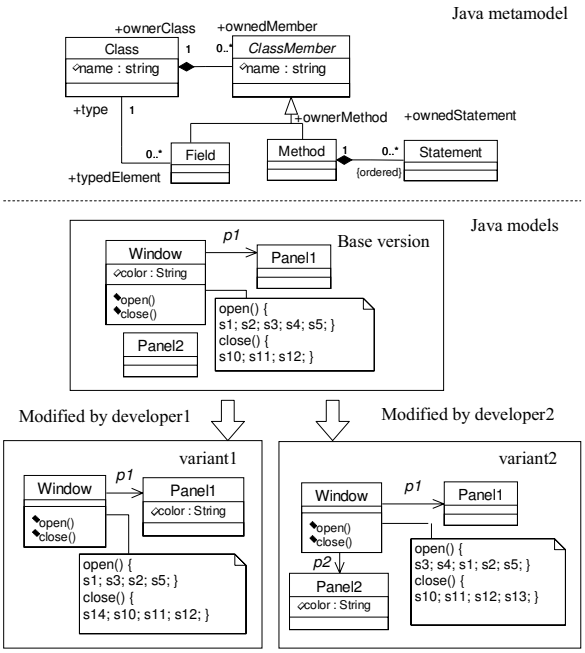

Fig 1. Java metamodel and three versions of Java models

3. MOF model merging: objectives

3.1. Delta calculation: comparing models

Extracting the modification from a model can be done in two manners. First, the tool that a developer uses to edit/modify models generates the log information of the modification he makes to the model [12]. Second, the modification is extracted by comparing the current version of the model and the version prior to the modification (*base version*) [13] [7]. The first way limits its application to tools that can write log. The second way does not have this limitation; hence, it can be applied with any tool. Therefore, we aim to propose a model comparison mechanism that inputs two model versions (base one and modified one) and extracts modifications from them.

According to MOF model structures, which consist of nodes, node contents (primitive property values) and links, the model comparison mechanism must identify the following changes to models: 1) creating/deleting nodes, 2) modifying node contents, and 3) inserting/removing links between nodes.

Moreover, MOF models can have ordered association ends. The value of an ordered association end is a sequence of node references. A tool can insert new sequence members, remove existing members or reorder existing members (i.e. inserting them to a new position in the sequence). Therefore, the mechanism must also detect changes to this order.

Extracting changes to a sequence means finding operations for transforming an old sequence and a new sequence. We require that change extraction be "optimal", i.e. the extracted operations must affect as few sequence members as possible, so that unaffected members can be concurrently modified.

For example, if a developer only moves one member in a sequence, these change operations are not optimal: "*all elements are removed and then reinserted with a new order*". The extracted operations would conflict with other concurrent modifications to this sequence, even if the latter does not affect the moved member. On the other hand, an optimal operation: "*move the specified element to a new position*", would allow other developers to concurrently manipulate other members than the moved one without conflicts.

3.2. Delta integration

3.2.1. Conflicts to be detected

Before integrating the calculated delta with the repository-side model, we need to detect and handle the conflicts that might occur if this repository-side model includes concurrent modifications. For example, a developer A and B concurrently edit their local copy of the same model, producing deltas d1 and d2 respectively. Supposing that A integrates his delta, before B, the first integration (by A) will cause no conflict as the repository observes no concurrent modification. On the other hand, the second integration (by B) can cause conflicts between d1 and d2, as B integrates d2 to the model that already includes d1.

The integration of conflicting deltas causes two problems. First, *lost update* is the problem that one delta forbids another delta's effects. Second, inconsistency is the fact that the integration of both deltas make the result model syntactically incorrect (i.e. not conforming to its metamodel). Our objective is to detect these problems so that developers can get aware of them and take actions for solving them.

We have studied the effect of applying delta operations (described in 3.1) concurrently. We identify the following cases that cause lost update.

- *Lost node content change* can be caused by two sub-cases. First, concurrent modifications to the same property of the same node result in one modification canceling another. Second, if a modification to a node's property is concurrent with the deletion of this node, then the modification will be lost.

- *Lost link creation* occurs when a link's creation is concurrent with the deletion of nodes representing the link's ends. Consequently, the link can not be created because its ends are missing.

- *Lost link order change* occurs when a link sequence (representing ordered association end value) is concurrently modified. This can happen in two sub-cases. First, the same sequence members are concurrently moved to different positions. Second, the same members are moved to a new position by one developer, and concurrently removed by another developer.

Solutions for detecting the first two problems (lost node content change, lost link creation) have already been proposed in [9] [11]. Therefore, in this work, we focus on detecting the third case, which concerns the specific characteristic of MOF models, i.e. ordered association end.

For the inconsistency problem, this work focus on detecting the inconsistency that is specific to MOF models: multiplicity violation. This checking can ensure several model characteristics expressed with multiplicity. First, it ensures that models are completed. For example, the Java metamodel forbids untyped fields by having the multiplicity of association end type (in metaclass Field) set to 1.

Moreover, it ensures the consistency of containment relationship, i.e., multiplicity can specify that a child node must have no more than one parent. For example, in the Java metamodel, the association end ownerClass (in metaclass ClassMember) is set to 1, which means that a class member must be owned by no more than one class.

3.2.2. Conflict resolution

In the case where conflicts are detected, a subset of conflicting operations must be dropped or altered. For example, if two developers concurrently move a Java field to two different classes, then one of the modifications must be dropped. Since the decision how to resolve conflicts depends on the developers' intent, a unique conflict resolution mechanism can not satisfy different requirements of each developer.

For this reason, we aim to offer a framework enabling conflict resolution programs to be built with little effort. To reduce this effort, we aim to offer the following key features in the framework:

1) Reporting the operations causing conflicts to the conflict resolution programs.

2) Enabling the programs to drop or alter conflicting operations.

These features enable programs to focus on the extracted information (i.e. in terms of conflicting operations), rather than letting programs directly manipulate raw information (i.e. models to be merged: base version and variants).

4. Our approach

4.1. Delta metamodel

Since our objective is to capture model modifications as deltas, first we need to define the delta representation. We propose the *delta metamodel,* shown in fig. 2, to represent the deltas themselves as models. A delta contains a set of modification operations, which are described by the following metaclasses.

- CreateNode /DeleteNode indicate the nodes that are created /deleted.

- ModifyPrimitive expresses the change to a primitive property: it specifies the new value of the property.

- ModifyLink expresses link insertion and removal. In other words, it expresses the change to the value of an association end (i.e. a reference set contained by a node). It specifies the insertion/ removal of node references to/from a reference set. This is represented by sub-metaclasses InsertLink and RemoveLink.

This metamodel can express changes to ordered association end values. InsertLink enables the insertion of references to both ordered and unordered reference sets. For an ordered link set (link sequence), the insertion position (positionAfter) needs to be specified in terms of the node reference to insert after (null value means the first position). E.g. inserting <d e> to <a b c> at postionAfter=a results in <a **d e** b c>.

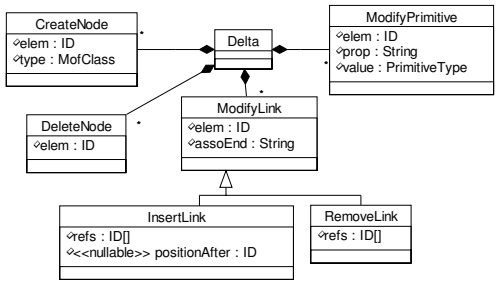

Fig 2. Delta metamodel

Compared to the XMI standard [17], which proposes the encoding of the delta between two MOF models, our approach offers an advantage of supporting concurrent modifications to the same reference sequence. In XMI, the expression of link insertion expresses with the absolute position (number index); therefore, when two concurrent modifications are applied to the same sequence at the same time, the index position of one of the modifications can become invalid [19]. Our approach uses relative positions, so it can avoid this problem.

For example, in the Java merging example, one developer inserts a statement s14 at the beginning of method close() and another inserts s13 at the end. By using relative positions, both modifications do not interfere with each other. On the other hand, by using indexes to specify insertion positions, inserting s14 before inserting s13 would make s13 be inserted to a wrong position.

4.2. Delta calculation

The delta calculation is a function that inputs a model variant (var) and its base version (base) and produces a delta model (conforming to the delta metamodel). Our mechanism uses node IDs to match nodes in two models in an accurate manner: if a node n1 in var has the same ID as the node n2 in base, then n1 has been copied from n2. The delta operations are produced with the following rules.

CreateNode/DeleteNode. The newly created nodes in var are the nodes that have no corresponding node in base. Similarly, the deleted nodes are the nodes in base that have no corresponding node in var.

ModifyPrimitive. Given two corresponding nodes in var and base, if their property values are different, then a ModifyPrimitive operation is generated.

ModifyLink. Changes to an association end value are extracted by comparing an old reference set with a new one. For an unordered association end, InsertLink and RemoveLink operations can be extracted by comparing the members of the two sets.

For an ordered association end, we also need to take into account changes to the sequence order. As stated the objectives, the change operations, which transform an old sequence to a new old, must be optimal. We use the following mechanism to extract these operations.

- Identifying removed members, similarly to the unordered case.

- Applying the Longest Common Subsequence algorithm [5] to find the members that are not affected by changes (i.e. the members in the common subsequence remain in the same order but the other members can be inserted between them).

- Examining members in the new sequence. The members that are not in the common subsequence either are new ones or have been moved. Therefore, InsertLink operations are generated for representing these changes. They refer to members in the common subsequence as insertion positions.

It is worth nothing that two ends of an association are redundant representation of the same information, e.g., a Field node contains a reference to its container Class node and vice-versa. To avoid this redundant information in the calculated delta, we choose to compare only one end of the association. If an association has an ordered end, then this end is chosen (so that changes to orders can be expressed). Otherwise, any of the two ends can be chosen.

Example. By applying our mechanism to extract modifications between each variant and the base version from the Java merging example, we obtain two delta models, d1 and d2, which contain the modifications in variant1 and variant2 respectively. We illustrate those models with a textual syntax, cf. the following code. In this syntax, each node ID, which allows the delta model to refer to a node, is underlined. Metaclasses' properties and association ends are written in *italic*. For example, CreateNode(s13, Statement) means that a Statement node with ID 's13' is created. InsertLink(close, *ownedStatement*, <s13> after s12) means that the node s13 is inserted in the ownedStatement association end value of the node close at the position after the node s12.

d1:

```
1.  //deleting Panel2
2.  DeleteNode(Panel2)
3.  // moving the field color from Window to Panel1
4.  RemoveLink(Window, ownedMember, <color>)
5.  InsertLink(Panel1, ownedMember, <color>)
6.  // moving s3 and deleting s4 in open()
7.  InsertLink(open, ownedStatement, <s3> after s1)
8.  RemoveLink(open, ownedStatement, <s4>)
9.  DeleteNode(s4)
10. // inserting s14 to close()
11. CreateNode(s14, Statement)
```

```
12. InsertLink(close, ownedStatement, <s14> @begin)
                                        d2:
1.  // moving the field color from Window to Panel2
2.  RemoveLink(Window, ownedMember, <color>)
3.  InsertLink(Panel2, ownedMember, <color>)
4.  // creating a field p2 and typing it with Panel2
5.  CreateNode(p2, Field)
6.  ModifyPrimitive(p2, name, "p2")
7.  InsertLink(p2, type, <Panel2>)
8.  InsertLink(Window, ownedMember, <p2>)
9.  // moving s3, s4 in open()
10. InsertLink(open, ownedStatement, <s3, s4> @begin)
11. // inserting s13 to close()
12. CreateNode(s13, Statement)
13. InsertLink(close, ownedStatement, <s13> after s12)
```

4.3. Conflict detection

Our conflict detection mechanism is a function that inputs two deltas (referred to as d1, d2) and returns the conflicts detected. It finds matching operations from both deltas (according to conflict detection rules) and reports them as a conflict. Therefore, a conflict detection rule is a condition for matching two groups of delta operations. As explained in 3.2.1, we focus, in this paper, on the detection of two problems: 1) lost link order change, and 2) multiplicity violation.

Detecting lost link order change. This problem is caused in two cases: conflict between two concurrent InsertLink operations, and conflict between InsertLink and RemoveLink operations. These conflicts can be detected by the following functions.

```
1.  Conflict detectLostLinkOrderChange(InsertLink i1, InsertLink i2) {
2.    if(i1 and i2 insert same references to different positions)
3.      return new Conflict(i1, i2);
4.    else return null;                }
5.  Conflict detectLostLinkOrderChange(InsertLink i, RemoveLink r) {
6.    if(i inserts references that are removed by r)
7.      return new Conflict(i, r);
8.    else return null;                }
```

By applying those functions to the Java merging example, we can detect two conflicts concerning the concurrent modifications to the open method's content. First conflict: s3 are inserted to different positions, i.e., InsertLink(open, ownedStatement, <s3> after s1) vs. InsertLink(open, ownedStatement, <s3, s4> @begin). Second conflict: in d2, <s3, s4> are moved to a new position but, in d1, s4 is removed from the method.

Please note that we do not consider as a conflict the case where RemoveLink deletes the reference represented by InsertLink's positionAfter. Applying the InsertLink operation before the RemoveLink operation can preserve the effect of both operations. For example, let <a b c d e f g> be the original sequence and an InsertLink operation aims to move g to the position after c while a RemoveLink operation aims to remove c. Applying the insertion before the removal yields the result <a b **g** d e f >, which preserves both modifications.

Detecting multiplicity violation. Multiplicity violation of a reference set (i.e. an association end value) can be detected by testing the application of all concurrent operations affecting this reference set. This test is performed on the reference set copied from the original model, rather than on the model itself, to avoid side-effects.

This detection is shown in the following algorithm. This algorithm inputs: 1) an association end value to be checked, which is specified by a tuple <n, e>, where n is a node and e is an association end (defined at the metamodel), and 2) two concurrent deltas (d1 and d2). It begins by identifying the operations, in each delta, that affect <n, e> (lines 3-10). Those operations are:

- The InsertLink and RemoveLink operations that directly modify <n, e> or indirectly modify <n, e> at the opposite end;
- The DeleteNode operations that prevent the InsertLink operations previously identified. I.e., they delete the nodes that are the ends of the links to be inserted.

Once all the affecting operations are identified, they are tested on the reference set copy (line 11-12). The InsertLink operations result in adding references to the set, while the RemoveLink and DeleteNode operations result in removing references from the set.

Next, the result is tested whether its cardinality is in the range specified by the multiplicity. If so, no conflict is return; otherwise, all the operations affecting <n, e> are reported as a conflict (lines 13-14).

```
1.  Conflict checkMultiplicity(Node n, AssociationEnd e,
2.                             Delta d1, Delta d2) {
3.    Set insertLinks1 = getInsertLinks (d1, n, e);
4.    Set affectingOps1 = insertLinks1
5.       .union(getRemoveLinks(d1, n, e))
6.       .union(getDeleteNodes(d1, insertLinks1));
7.    Set insertLinks2 = getInsertLinks (d2, n, e);
8.    Set affectingOps2 = insertLinks2
9.       .union(getRemoveLinks(d2, n, e))
10.      .union(getDeleteNodes(d2, insertLinks2));
11.   Set value = getValueCopy(n, e);
12.   apply(n, e, affectingOps1.union(affectingOps2));
13.   if( multiplicityOK(ae, value) ) return null;
14.   else return new Conflict(affectingOps1, affectingOps2);  }
```

This algorithm can detect the following conflicts in the Java merging example:

- *Model incompleteness*. One developer removes the class Panel2, while another creates a field (p2) and types it by this class, the deletion result in this field being untyped, which violates type's multiplicity (exactly one). The detected conflict reports conflicting operations: DeleteNode(Panel2) from d1 vs. InsertLink(p2, type, <Panel2>) from d2.

- *Containment relationship violation*. Two developers put the field color to two different classes, applying both modifications causes the field to be contained by two classes, which violates ownerClass's multiplicity (exactly one). The detected conflict reports conflicting operations: RemoveLink(Window, ownedMember, <color>), InsertLink(Panel2, ownedMember, <color>) from d1 vs. RemoveLink(Window, ownedMember, <color>), InsertLink(Panel1, ownedMember, <color>) from d2.

4.4. Flexible conflict resolution framework

Our framework enables a conflict resolution program to be realized as a function having the following parameters:

- A set of detected conflicts. This parameter enables the program to examine each detected conflict and to resolve it.

- Two deltas (d1 and d2) to be merged. These parameters enable the program to drop or alter the operations in those deltas.

This framework offers the following programming facilities:

1) An API for examining and modifying delta operations. As deltas themselves are represented as models, manipulating deltas is done in the same way as models.

2) An API for examining conflicts. It enables a program to obtain a conflict description (e.g. lost link order change, multiplicity violation) and the delta operations from the d1 and d2 sides that involve the conflict.

3) A conflict detection function to recalculate the conflicts. This function refreshes the conflicts that still remain, once both deltas have been altered.

The following code shows how our framework APIs can be used by conflict resolution programs. We show two functions that can apply two different policies to resolve conflicts:

P1: The latest delta integrated (i.e. d2) is priority. If conflicts occur, the operations in this delta will be chosen, and the conflicting ones in the other delta will be dropped. This mechanism is showed in line 1-3.

P2: Node deletions are less priority. This policy aims to avoid lost information. It can only solve a conflict involving a node deletion at one side. It drops the DeleteNode operation causing conflict, in order to keep the modifications by the other operations, cf. lines 4-9.

```
1.   void resolve_p1(Conflict c, Delta d1, Delta d2) {
2.     d1.drops( c.getOpsFromD1Side() );
3.   }
4.   void resolve_p2(Conflict c, Delta d1, Delta d2) {
5.     if( isDeleteNode(c.getOpsFromD1Side()) ) {
6.       d1.drops (c.getOpsFromD1Side() );
7.     } else if( isDeleteNode(c.getOpsFromD1Side()) ) {
8.       d2.drops(c.getOpsFromD2Side() );
9.     } else { throw "cannot resolve" }
10.  }
```

4.5. Implementation and application for CASE tool interoperability

We have applied this merging mechanism in ModelBus, a platform for CASE tool interoperability. ModelBus enables several developers to use different CASE tools to realize a collaborative development. I.e. they can concurrently use different tools to update the shared models. The ModelBus architecture for supporting this collaboration is composed of the following components.

Repository. We reuse the basic functionalities of existing Software Configuration Management (SCM) repositories (e.g. CVS or Subversion [21]) for storing models that are shared among several tools and for keeping track of the modification history on those models. In our approach, shared models are stored in a repository as XMI files [17]. Therefore, any file-based repository can be used for storing XMI files and the modification history on those files. In our ModelBus prototype, we have chosen the CVS repository, which is widely used in practice.

This repository enables several tools to modify shared models as follows. A tool (T1) can *check out* a model (an XMI file) from the repository, then modify it locally, and finally *commit* the new version of this model to the repository for replacing the previous version. In this committing process, the repository checks whether the model has been concurrently modified by another tool (i.e., the other tool has committed its version before). If so, then the repository informs T1 that it needs to merge its modification with the other tool's modification, before committing.

In this case, the tool will perform the merging mechanism that we propose. To do so, the tool needs three versions of the model (base version and two variants). One of the two variants is the tool's *local version*. Since the repository keeps track of modification history on models, it can provide the tool with the base version (in the case that the tool has not kept it since last check-out) and the other variant, i.e. the *repository version*, which contains the concurrent modification of another tool. Our merging mechanism will produce the *unified version,* which includes the modifications in both variants. The tool can then commit this unified version to the repository.

Please note that, in the case that more than two tools perform concurrent modification on the same model,

each tool can do merging/committing one by one, in order to accumulate the modifications of all tools.

Merging framework. This component contains the implementation of our merging mechanism. In this implementation, we use the Eclipse Modeling Framework (EMF) technology [4] for manipulating models. EMF offers an API for allowing model nodes to be manipulated (created, deleted and modified) in the same way as Java objects. Based on this API, we have implemented the delta calculation, conflict detection and delta integration mechanisms. We use the reflective programming technique, which enable our framework to manipulate models whose structures are unknown in advances (i.e. their metamodels are unknown). For this reason, our framework supports the merging of any kinds of models (UML and Domain Specific Models).

The CASE tool can use our merging framework as follows. The framework offers to CASE tools the following API for starting the merging mechanism:

```
MergeModel merge(InputStream base, InputStream variant1, InputStream variant2)
```

This operation inputs three model versions (base, variant1, variant2) encoded in the XMI format, and produces the result (MergeModel object) which informs whether or not conflicts occur. If no conflict occurs, then the MergeModel object (the result) will provide the tool with the unified version.

In the case that conflicts are detected, our framework enables tools to solve this problem with any conflict resolution programs that can apply different conflict resolution policies (described in 4.4). To do so, the tool sends MergeModel object the conflict resolution program by called the following interface operation, which is expected to be implemented by all conflict resolution programs:

```
void resolveConflict(MergeModel mergeModel, OutputStream unifiedVersion)
```

A conflict resolution program can obtain the conflict information, the deltas, and the base model from the MergeModel object in order to perform its conflict resolution logics and produce the unified version. Depending on its logics, it can either solve the conflicts in a completely automatic way or ask for the developer's decisions. We have implemented two sample conflict resolution programs that any tools can use (if they do not have their customized ones). The first one uses the policy of giving the priority the latest delta integrated. It performs conflict resolution in a completely automatic way.

The second one is developer-interactive. It offers a simple GUI showing the deltas and the conflicts to the developer, and allowing the developer to select the conflicting delta operations to be dropped or altered. In an example of this GUI (fig. 3), the left part (*local diff*) shows the modification in the local version (compared with the base version), and the right part (*remote diff*) shows the modification in the repository version. When a node in the left or right part is selected, the bottom part (*properties*) will show the details of modifications to this node and the related conflicts. This GUI enables the developer to locate the conflicting delta operations in the local and repository versions, and to selectively drop or alter them.

Management of node IDs. Our merging mechanism requires the existence of node IDs. Therefore, we require that the XMI files in the repository contain not only model information but also node IDs. However, not all CASE tools preserve node IDs. Node IDs can be lost when those tools load models from XMI files to an object representation (for manipulating them) and save them back to files. For this reason, ModelBus proposes an ID management mechanism, which allows tools to load, save and manipulate models without worrying about ID preservation. Our previous work [18] presents an approach to assign IDs to nodes and to preserve them when loading and saving models in a tool-transparent way. This approach consists in providing tools with the loadModel and saveModel operations which mask the ID management mechanism from tools. The loadModel operation creates an in-memory table associating node objects with their IDs obtained from the loaded XMI file. The saveModel operation saves the IDs in this table together with the model to an XMI file. This operation is also responsible for assigning new IDs to newly created nodes.

So far we have implemented the loadModel and saveModel operations for converting between XMI and the EMF object representation, which is becoming a popular object representation for new generation CASE tools. The same principle can be used for realizing those operations for other object representations.

5. Related works

Delta calculation. As explained, model comparison requires matching nodes in two model versions. Rather than using node IDs, an alterative way to match nodes is to observe node similarity, e.g. comparison of UML class diagrams [23], XML files [22], and Abstract Syntax Tree [6] [24] [3]. In these approaches, two nodes that have similar contents and similar neighbor nodes are matched. These approaches support delta

extraction even if node IDs are unavailable. However, their result is only the estimation of matching and can contain errors (error rate study is reported in [23]).

Lindholm has proposed a XML document merging mechanism [7], which shares the two points with our work. First, it considers changes to the children's order in a parent XML node. Second, it ensures that a node has exactly one parent (except root). Our work is more generic. First, the ordered relation in MOF is not necessarily parent-child. Second, it supports any kind of multiplicity (not only parent-child constraint).

Conflict detection & resolution. A delta integration mechanism for graphs has been proposed in [12]. Its uses a predefined conflict detection mechanism: The operations are applied according to the order of delta integration (e.g. d1 then d2). An operation can be applied only if its precondition holds; otherwise it is dropped. The applied operation can cancel the effects of previous operations. The authors did not propose the way to prevent lost update problems. In our work, we offer a mechanism to detect the problems and enable developers to solve them automatically and with flexibility.

Munson & Dewan has suggested the use for merge matrix as a conflict detection and resolution framework [11]. This matrix indicates a pair of operations that are conflicting and the resolution action. On the other hand, our mechanism can detect a conflict that involves more than two operations (e.g. multiplicity conflict). We also suggest representing deltas as models to facilitate their manipulation (analyzing, dropping or altering them).

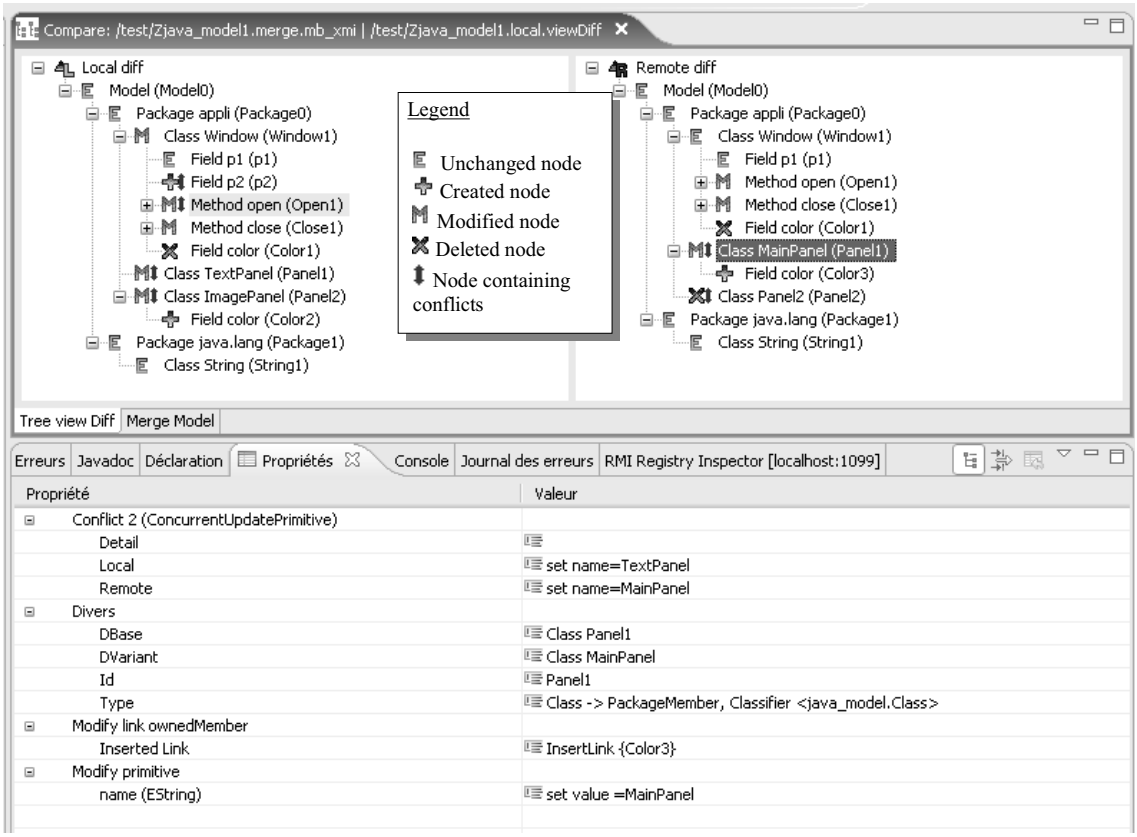

Fig 3. A simple interactive conflict resolution program

6. Conclusion and future works

We presented a mechanism to merge MOF models, main software artifacts in the MDA approach. This mechanism takes into account MOF features (ordered association end and multiplicity). Its contribution is to support collaboration in an open tool integration environment.

For future works, we consider the following improvements:

- *Extending consistency check*. Besides multiplicity, MOF enables metamodel designers to define model-specific constraints in an expressive way with Object Constraint Language (OCL) [16]. These constraints can express that, for example, in a Java model, method call expressions must have valid arguments (must conform to method signatures). We consider extending our conflict detection mechanism to ensure that the merge result is consistent as regards these constraints.

- *Visual support for conflict resolution*. Not all conflicts can be automatically resolved by programs. Developers may require the ability to visualize them for reflecting on what to do. In this work, we have proposed a simple user interface enabling developers to do so. However, this user interface does not take into account the dedicated notation of each model kind (UML diagram notations or Domain Specific Model notations), which is more intuitive to developers. In future works, we would like to study about how to provide a more user-friendly interface for supporting conflict resolution. This interface should be adaptable to different model notations. We will also consider integration of semi-automated conflict resolution programs with this user interface.

7. References

[1] Adams, E.W., Honda, M. and Miller, T.C., Object Management in a CASE Environment, *Proc. of the 11th Int'l Conf. on Software Engineering*, 1989.

[2] Blanc, X., Gervais, M-P. and Sriplakich, P., Model Bus: Towards the Interoperability of Modeling Tools, *Proc. of the European MDA Workshop: Foundations and Applications*, LNCS 3599, 2005.

[3] Buffenbarger J., Syntactic Software Merging, *Proc. of the Software Configuration Management Workshop: selected papers*, LNCS 1005, 1995.

[4] Eclipse, Eclipse Modeling Framework Project, http://www.eclipse.org/emf/.

[5] Hirschberg D. S., Algorithms for the Longest Common Subsequence Problem, *J. of the ACM,* 24(4), 1977.

[6] Hunt, J.J. and Tichy, W.F., Extensible Language-Aware Merging, *Proc. of ICSM'02*, 2002.

[7] Lindholm, T., A Threeway Merge for XML Documents, *Proc. of the 2004 ACM Sym. on Document Engineering*, 2004.

[8] Lippe, E., and van Oosterom N., Operation Based Merging, *Proc. of the 5th ACM SIGSOFT Sym. on Software Development Environments*, 1992.

[9] Mens, T., Conditional Graph Rewriting as a Domain-Independent Formalism for Software Evolution, *Proc. of the Int'l Workshop AGTIVE'99*, LNCS 1779, 2000.

[10] Mens, T., A state-of-the-art survey on software merging, *IEEE Trans. on Software Engineering*, 28(5), 2002.

[11] Munson J.P. and Dewan P., A Flexible Object Merging Framework, *Proc. of the 1994 ACM Conf. on Computer Supported Cooperative Work*, 1994.

[12] Oda, T. and Saeki, M., Generative Technique for Version Control Systems for Software Diagrams, *Proc. of ICSM'05*, 2005.

[13] Ohst, D., Welle, M., and Kelter, U., Difference Tools for Analysis and Design Documents, *Proc. of ICSM'03*, 2003.

[14] OMG, *MDA Guide Version 1.0.1*, document no: omg/2003-06-01, 2003.

[15] OMG, *Meta Object Facility version 2.0*, document no: formal/06-01-01, 2006.

[16] OMG, *Object Constraint Language (OCL) Specification version 2.0*, document no: ptc/2005-06-06, 2005.

[17] OMG, *XML Metadata Interchange (XMI) Specification version 2.0*, document no: formal/03-05-02, 2003.

[18] Sriplakich, P., Blanc, X. and Gervais, M.P., Supporting transparent model update in distributed CASE tool integration, *Proc. of the ACM Sym. on Applied Computing*, 2006.

[19] Sun, C. et al., Achieving Convergence, Causality Preservation and Intention Preservation in Real-Time Cooperative Editing Systems, *ACM Trans. on Computer-Human Interaction,* 5(1), 1998.

[20] Tichy, W.F., RCS - A System for Version Control, *Software Practice and Experience*, 15(7): 637-654, 1985.

[21] Tigris, Subversion project, http://subversion.tigris.org.

[22] Wang Y., Dewitt D.J. and Cai J-Y., X-Diff: An Effective Change Detection Algorithm for XML Documents, *Proc. of 19th IEEE Int'l Conf. on Data Engineering*, 2003.

[23] Xing, Z. and Eleni, S., UMLDiff: an Algorithm for object-oriented design differencing, *Proc. of the 20th IEEE/ACM Int'l Conf. on Automated Software Engineering*, 2005.

[24] Yang. W, Identifying Syntactic Differences Between Two Programs, *Software Practice and Experience*, 21(7), 1991.