Collaborative Software Engineering on Large-scale models: Requirements and Experience in ModelBus*

Prawee Sriplakich[1,2], Xavier Blanc[2], Marie-Pierre Gervais[2]

[1]INRIA Futurs
Villeneuve d'Ascq, France
Prawee.Sriplakich@inria.fr

[2]Laboratoire d'Informatique de Paris 6 (LIP6)
Paris, France
{Xavier.Blanc, Marie-Pierre.Gervais}@lip.fr

ABSTRACT

This work presents an approach for realizing Model-Driven software engineering in the distributed and multi-developers context. It particularly focuses on the scalability problems in a complex software project involving a large set of inter-connected models: (1) how to manipulate large data volume with limited computing resources, and (2) how to maintain consistency of inter-model links in a large model set, facing to concurrent model updates. As a solution, we propose the *scalable copy-modify-merge mechanism*, which allows each developer to copy only a model subset from the entire model set, to manipulate this subset locally, and to merge it back to the repository. This mechanism ensures the global consistency of the model set, particularly against dangling links. Our approach is generic: it is applicable to all model types (UML and Domain-Specific Models). Also, it offers interoperability with existing, heterogeneous CASE tools. Its prototype implementation in the ModelBus environment is now available on the Eclipse project "MDDi".

1. Introduction

Supporting collaboration among multiple developers in a distributed system is an essential requirement in complex software projects. In such as collaboration, each developer, located in a geographically distributed environment, needs to manipulate (create, analyze, and update) software artifacts that are shared among the team's members. A natural way for realizing this collaboration is to use a distributed environment offering the copy-modify-merge functionalities [1] – a *collaborative environment*. In this environment, shared software artifacts are stored in a server called the *repository*. Each developer copies those artifacts from the repository to his (her) *workspace* (local machine). Then he can manipulate them independently from the other developers. Each developer can share the updates locally made in his workspace with his colleagues by merging the workspace with the repository. Therefore, the result seems as if all team members are working on single-copy and highly-available models.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'08, March 16-20, 2008, Fortaleza, Ceará, Brazil.
Copyright 2008 ACM 978-1-59593-753-7/08/0003…$5.00.

Today, software engineering techniques are shifting to Model-Driven Engineering (MDE), where models are the main software artifacts. However, we identify the following difficulties in the realizing a collaborative environment for MDE (an *MDE environment*):

Scalability. Complex software engineering requires the use of multiple models. Each model describes a particular software module at a particular viewpoint. For example, in UML, use-case models, interaction models, and class models can be used to define, respectively, functional, behavioral, and structural views of the same system. Moreover, those models are interconnected: the links among models (*inter-model links*) represent relations among views, relations among software modules [19], and traceability relations (e.g. links between user requirements and designs [17]). Therefore, a set of models describing a system can be considered as a large-scale and complex data structure, which we call a *modelbase*.

The size of a modelbase grows according to the system's complexity. Let's consider an example of a large system containing 25,000 classes presented by [7]. Supposing that, all model elements for describing all views of each class (e.g. structural, behavioral, test views) take 100 KB in average, we estimate that all the models that completely describe this system will be as large as 2.5 GB. However, it is difficult to manipulate entirely the large modelbase at a time with limited computing resources. Therefore, we identify the need to partition a large modelbase into different parts, so that each part (each model) can be manipulated separately.

Link consistency. Modelbase partitioning and the fact that each model can be manipulated separately lead to the need to preserve inter-model links, because those links represent important software engineering information, including links among views, links among software modules, and traceability. However, concurrent updates can cause inconsistency regarding inter-model links (dangling links). Moreover, facing to a large data volume, preserving link consistency becomes more complex [12].

Besides scalability and consistency, we also consider the following requirements, which are important for usability of the MDE environment:

Genericity. Different models involved in a complex software project can be expressed with different metamodels, i.e. they are

* This work has been partially supported by the ModelPlex European integrated project FP6-IP 034081 (Modeling Solutions for Complex Systems).

of different *types*. Besides using UML, which proposes several model types (use-case models, class models, interaction models, etc.), developers can require using other model types, based on domain-specific metamodels. Therefore, the MDE environment must be able manage models expressed with any metamodel (both UML models and Domain-Specific Models), not to limit developers to a single model type.

Interoperability with existing CASE tools. Both academic and industrial CASE tools offer different functionalities, such as model transformation [3], model checking [18], and code generation [4]. The MDE environment should offer developers the freedom in using any CASE tools adapted to their projects. Therefore, an open architecture allowing existing CASE tools to be integrated to this environment is required.

The contributions of this work are summarized into the following points:

1) We identify essential functionalities that an MDE environment should provide to support collaborative software engineering.

2) We present the state of the art of the existing techniques in object management and software merging domains to identify their lacking points and the extensions needed.

3) We present our experience in realizing those functionalities in a MDE environment called ModelBus. The ModelBus implementation is available in the Eclipse project MDDi (Model Driven Development integration; http://www.eclipse.org/mddi). This implementation has been validated in the European project ModelWare (http://modelware-ist.org).

This article is organized as follows: Section 2 presents in more detail the problems of applying the copy-modify-merge paradigm to large-scale models. Section 3 surveys related works on the identified problems. Part 4 presents our experience in realizing ModelBus, before a conclusion.

2. MDE Environment: Requirements

Before explaining the requirements, we first present an overview of the copy-modify-merge paradigm. This paradigm involves three main concepts: *repository-workspace*, *delta extraction*, and *delta integration*. **Repository-workspace** is a system composed of a repository and workspaces. The repository serves for storing shared models. It must also offer *version control* functionality for storing model update history, in the form of model versions. This functionality is necessary for model merging (c.f. next paragraph), and for the software evolution analysis along all software project phases [25]. A workspace stores a model copy at the developer side. It allows a developer or a tool to *retrieve* models from the repository, and to *commit* a new model version back to the repository.

Delta extraction and **delta integration** are parts of the merging mechanism, which allows each developer to integrate his updates contained in his workspace's models (Vlocal version) to the repository's current version (Vrep), which, as a consequence of collaborative work, can contain the concurrent updates committed by other developers (cf. next figure). Delta extraction consists of extracting the updates (i.e. *deltas*) contained in the Vlocal version, by comparing this version with the model version that has been initially copied from the repository (Vbase). A delta contains a set of *update commands*, which express how Vbase evolves to Vlocal. An update command can express model element creation/ deletion, and modification to an element's attributes, including modification to its *primitive attribute*, and modification to its *link attribute* (an attribute containing links to elements).

Delta integration consists of applying the update commands, contained in the deltas, to Vrep. This integration will produce the merged version (Vmerged) containing both the updates made by this developer and the ones concurrently made by other developers.

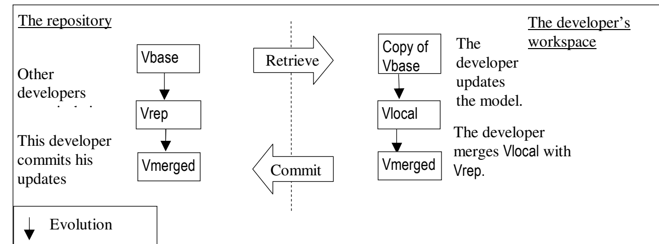

Figure 1. Different model versions involved in the merging mechanism

2.1 Requirements on the repository-workspace system

Scalability. Although a modelbase, which contains a set of interconnected models, can be of large size (in the order of GB), a developer usually focuses a particular software module/ viewpoint at a time; hence he needs to manipulate only the model subset corresponding to this module/ viewpoint. Therefore, the repository-workspace system should offer functionality **to partition a modelbase** into several models, and should enable a developer to retrieve the model subset that he wants to manipulate, and to commit the model subset that he has updated.

Once a developer retrieved models from the repository, he needs to use a variety of tools to manipulate (analyze, update) the retrieved models. Therefore, it is necessary to offer scalability in the way tools manipulate large-scale models. In fact, before manipulating models, a tool needs to load the models into memory, i.e. constructing data structures in memory for representing the models. However, we experience a problem that, for complex software, the models in the developer's workspace (on disk) may be too large to be entirely loaded into memory. Therefore, a tool needs the ability to load only a subset of models to memory: **scalable model loading**. The difficulty in realizing this functionality concerns the management of inter-model links: The loading mechanism must deal with links from loaded models towards unloaded models. Moreover, the unloaded models may be unavailable in the developer's workspace (since the developer may have retrieved only a model subset from the repository); therefore, the mechanism must also deal with links to unavailable models. As tool programmers require transparency to this difficulty, this loading functionality should be provided by the workspace.

Interoperability with existing CASE tools. Different tools provided by different organizations can be implemented with heterogeneous technology. They can be written in different programming languages, and use different model representations (e.g. Java, C++ objects). For interoperability with those heterogeneous tools, the workspace must enable the tools to read/

update the retrieved models, regardless of their internal model representations, which can be heterogeneous.

2.2 Requirements on delta extraction

- **Genericity**. The delta extraction principle consists of matching elements in two model versions (Vbase and Vlocal). For each pair of matching elements, their contents are compared to generate update commands. For elements with no matching, the ones in Vlocal are considered to be created elements, while the ones in Vbase are considered to be deleted.

Element matching has been recognized as a difficult mechanism in delta extraction [24]. It needs to identify matches among multiple possibilities, which grow rapidly according to the number of model elements. Concerning this matching, genericity is required to support all model types. Therefore, the mechanism must be able to match any model elements, which are instances of any metaclass, even though those elements have different attributes.

Scalability. We identify that, for scalability, partitioning a modelbase into several models is necessary, because it enables a developer to restrict delta extraction to a selected model subset. However, this partitioning creates a new problem: the detection of element moving among models, which is an essential action for reorganizing software modules and views. Moving an element from one model to another model reflects the moving of this element to a new software module/ view.

Moving an element is different from deleting this element from an source model, and creating an identical model element in the target model: in the case of moving, all links to the moved element must remain valid. However, this element moving would be undetected, if the delta extraction mechanism analyzed only the source model or only the target model. Therefore, our requirement concerns not only scalability (delta extraction on a model subset), but also the model reorganization support (element moving detection).

2.3 Requirements on delta integration

The delta integration mechanism consists of applying update commands (contained in deltas) to the repository's model version (Vrep). This mechanism needs to be aware of the conflicts between the update commands to be applied and the concurrent updates previously integrated by other developers.

In this work, we focus on the following conflicts that dedicated to large-scale models.

2.3.1 Conflicts caused by element moving

Element moving among models is a frequently-used operation for reorganizing or re-architecting software specification. For instance, to re-architect class models, developers may move classes among packages. However, facing to concurrent updates, element moving can interfere with update commands that target the moved elements in the following cases.

Updating moved elements. Let us consider the application of update command U, which expresses an update to element E at model M1. However, before the command is applied, another developer has moved E to another model M2. Thus, this moving will interfere with update command U, because, in the repository, E is missing in model M1.

Moving referenced elements. Let us consider that a developer intends to apply an element moving to the repository. Since the developer works on a part of the modelbase, he can be unaware of the links from the rest of the models to the element to be moved, which will become dangling.

2.3.2 Conflicts caused by element deletion

Element deletion also interferes with update commands that target the deleted elements in the following cases.

Creating links to deleted elements. Let us consider the application of update command U, which expresses the link creation to element E. However, E might have been deleted by other developers. Thus, the created links will become dangling.

Deleting referenced elements. Let us consider that a developer intends to apply an element deletion to the repository. Since the developer works on a part of the modelbase, he can be unaware of the links from the rest of the models to the deleted element, which will become dangling.

We note that the fact that a developer manipulates only a part of the modelbase adds difficulties to the all presented problems. If a developer updates only a part of the modelbase, he will concentrate on merging the updated part. However, the merging of this part requires the analysis of the rest of the modelbase, which may contain elements related to the updated ones, for detecting/ solving the presented conflicts.

Moreover, this analysis should avoid exhaustive element searching for scalability. For example, the mechanism to deal with problem "updating moved elements" should avoid searching exhaustively the move target locations; similarly, the mechanism to deal with "deleting referenced elements" should avoid searching exhaustively the links towards the elements to be deleted.

3. Related works

3.1 Repository-workspace

OODB systems. Several works, such as [15] [16], propose an Object-Oriented Database (OODB) system for managing storage and version control for software artifacts. In those works, model elements are considered to be objects managed by an OODB system. CASE tools behave as client programs: they access (read and write) those objects through an OODB *client API*, which offers operations for retrieving objects from the server to the workspace, manipulating (reading, writing) them, and committing them back to the server.

On the other hand, our goal consists of building an MDE environment that enables developers to use heterogeneous tools to perform collaborative work. We identify that OODB systems do not correspond to this goal for two reasons: (1) An OODB client API is usually complex. Thus, to integrate a tool with an OODB system, the tool programmers must spend considerable effort in writing code against this API (up to 30% of programs [23]). (2) The client API imposes an object representation to be used by the client programs. On the other hand, existing tools are already coupled with particular object representations, e.g. JMI (http://-java.sun.com/products/jmi), EMF (http://www.eclipse.org/emf), or proprietary representations (C++ objects [8]). Therefore, tool programmers would suffer the impedance mismatch problem [2] when adapting their tools with an OODB system.

File-based versioning systems. Another approach for managing model storage and version control consists of using file versioning systems, such as CVS, or Subversion. By adapting those systems, a modelbase can be partitioned into a set of files, for scalability in model manipulation.

There exist works on adapting a file versioning system for collaborative model manipulation (e.g. [9]); however, few works focus on the large-scale aspect of models. In our experience, when multiple tools are used together in collaborative work, one tool usually faces the problem on resolving inter-model links encoded by other tools. Moreover, tool programmers also face difficulties in how to make the tool manipulate a model subset, without destroying links between this model subset and the rest of the modelbase.

We identify that, between the two approaches, file versioning systems are more appropriate for a MDE environment in terms of interoperability with existing tools. A file versioning system offers loose coupling with heterogeneous tools; i.e., it offers a simple way to convert between the models and the tools' internal object representations (e.g. Java, C++ objects). Moreover, models can be encoded as files with the XMI standard, which now supported by a wide variety of tools. This standard simplifies the coupling of tools with a file versioning system.

3.2 Delta extraction

Update interception. By exploring existing works, we identify three main approaches on delta extraction. The first approach is based on update interception [13]. In this approach, when tools update models, the objects representing models are expected to generate model update events, which will be recorded as update commands. This approach has restriction on model representations; therefore, it is not suitable for integrating heterogeneous tools.

Model comparison based on element similarity. The second approach consists in comparing two model versions. In this, two elements in two model versions are matched, their contents compared, then update commands are generated from this comparison. Element matching is based on their similarity. To apply this approach, heuristic matching rules need to be defined for each type of data elements. e.g., Xing [24] proposes matching rules for different types of elements in UML class models, such as: two classes match if they have similar names and similar properties; two operations match if they have similar signature. Wang [22] proposes matching rules for XML documents, which are restricted to tree data structures (graph comparison is not supported). For this reason, this approach does not correspond to our requirement on genericity: it is not applicable to all model types. Moreover, this approach can produce false matching (false negative and false positive), because it uses heuristic rules (Xing [24] shows an error rate study).

Model comparison based of IDs. The third approach uses IDs for matching model elements (e.g. [14] [9]). Each element owns a persistent ID; therefore, two elements with the same ID from two model versions match.

According to our study, model comparison based on ID offers a good promise for its application in an MDE environment. First, it offers genericity: it can match all kinds of model elements (UML and DSM models). Moreover, unlike the approach based on element similarity, the ID-based approach enables accurate detection of element matching. However, to apply this approach it is necessary to deal with ID management, facing heterogeneous tools.

3.3 Delta integration

The survey [10] presents several works on delta integration. Those works propose the detection of different conflict kinds, including conflicts causing lost update; conflicts causing syntax or semantic inconsistency. In the MDE context, a generic mechanism for detecting lost update conflicts and syntax inconsistency conflicts (non-conformance to metamodels) have already been proposed in [20]. On the other hand, we do not consider semantic conflicts, because they depend on the semantics of each model type.

To our knowledge, the large-scale aspect of conflict detection has not yet been addressed. In fact, existing conflict detection mechanisms operate on an entire data structure. Those mechanisms are not appropriate for a modelbase, which is too large to be manipulated entirely at a time. On the other hand, an appropriate mechanism should operate only on a specified model subset, while preserving link consistency of the entire modelbase.

Concerning conflict resolution, Munson [11] proposes a framework allowing developers to program their customized conflict resolution rules. This approach is powerful, but it is only suitable for experts. Grundy [6] proposes an interactive conflict resolution program, which enables developers to accept or reject conflicting commands causing conflicts. This approach offers good control to developers; however, it is time-consuming. On the other hand, Oda [13] proposes default automated conflict resolution rules, which consist in accepting update commands if their preconditions hold, or rejecting them otherwise. This approach offers less control to developers, but it is automated.

We propose that the use of manual and automated conflict resolutions should be combined; i.e., an MDE environment should offer flexibility to developers in choosing the way to resolve conflicts, either manually or automatically.

3.4 Summary

This section surveyed several works related to repository-workspace systems, delta extraction, and delta integration. This survey revealed the concepts that can be reused, i.e. the use of file-based versioning systems for interoperability with heterogeneous tools; the use of IDs in delta extraction for achieving genericity; and the use of existing conflict detection rules in delta integration. However, we also identified lacking points of those works – ability to manage large-scale, interconnected models. More precisely, few works focus on the management of model partitioning and inter-model links (e.g. detecting element moving among models, then repairing references towards them). To our knowledge, recent CASE tools supporting collaborative model edition, such as IBM Rational Software Architect or MagicDraw, do not handle those problems.

4. Experience in ModelBus

This section reports our experience in realizing the ModelBus environment. Only core concepts of ModelBus are presented here (please refer to the Ph.D. thesis [21] for more details).

ModelBus reuses a file versioning system (specifically CVS) for storing models. It allows each developer to retrieve models from the CVS repository to his workspace (cf. fig. 2, step 1), then to

use any tools to manipulate the retrieved models (step 2), and finally to merge and commit the models to the repository (step 3). ModelBus offers two components for supporting those actions: *WorkspaceManager* and *Tool Adapter*.

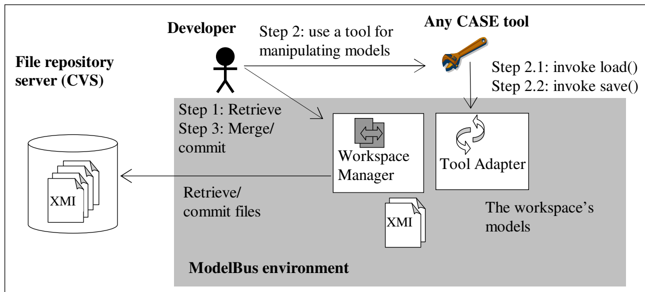

Figure 2. Overview of ModelBus

WorkspaceManager enables the developer to retrieve models from the repository, to merge the models, and to commit them back to the repository. It extends a CVS client with model merging functionalities (delta extraction and delta integration). This component can interoperate with any CASE tools used by the developer by sharing XMI files with them.

Tool Adapter allows a CASE tool to be integrated with the ModelBus environment: It enables the tool to manipulate the workspace's models transparently to model encoding. In particular, it offers functionality for converting models stored as XMI files in the workspace to objects in memory, according to the tool's model representation.

4.1 Extension to the file versioning system

Model partitioning and link encoding. In ModelBus, each model is stored in the repository and in workspaces as an XMI file. A model is identified with the file path, which is relative to the repository's root directory (e.g. /module1/model1.xml). Each model element is encoded as an XML element. It owns a persistent ID, which is stored in XML attribute xmi:id. Its content (sub XML elements, or attributes) can include links towards other model elements. A link is encoded in the form: *the target model's file path # the linked element's ID*. If the linked element is in the same model, the target model's file path can be omitted.

Our contribution consists of proposing a specific use of XMI with the following characteristics: (1) It ensures the availability of element IDs, which are necessary for model merging. (2) It formalizes the encoding of links between models to avoid the ambiguity in link resolution (when tools try to navigate through the links). We propose that the directory structure of the models in the developer's workspace remain the same as the one in the repository (e.g. if R is the root directory of the developer's workspace, then the file /module1/model1.xml in the repository must be placed at the R/module1/model1.xml in the workspace). Therefore, the links encoded according to our approach can be resolved without ambiguity in all developers' workspaces. (3) Our addressing scheme is suitable for large-scale models (using model path and ID). It enables efficient link resolution without exhaustive search of linked elements in the entire modelbase.

Scalable model loading. Tool Adapter offers load and save operations for enabling tools to manipulate models in the workspace. Its added values consist in (1) masking the ID management from the tool, (2) offering the scalable model loading mechanism for scalability. This mechanism applies the previously described link encoding and resolution convention.

ID management consists of preserving IDs when a model is loaded in memory, and then saved back. Tool Adapter uses an *ID table* for maintaining the associations between IDs and the *model objects*, which represent model elements in memory. The ID table management is transparent to tools. The ID table contains entries <ID, the pointer to the model object>. When a model is loaded, Tool Adapter inserts entries to the ID table. During model saving, Tool Adapter writes those IDs back together with the model. For the new model elements, Tool Adapter automatically assigns IDs to them. It generates IDs with the UUID (Universally Unique Identifier) standard, which ensures ID uniqueness.

Concerning ID management, our contribution concerns the following points: (1) Tool Adapter ensures the transparency of ID management regarding tools. (2) Our approach is applicable to all model representations, as the ID table entries can point to any objects (Java, C++, etc). (3) It offers scalability in terms of memory usage. The ID table's size changes according to the models currently loaded in memory. We apply the weak pointer concept [5], so that, when models are unloaded from memory, the corresponding ID table's entries will be automatically removed together with the models.

Concerning scalable model loading, Tool Adapter enables a tool to load a set of interconnected models in the on-demand manner. Its load operation enables the tool to select the models to be initially loaded to memory. It returns a set of corresponding model objects to the tool. The tool can manipulate those objects, and navigates through their links. When the tool attempts to navigate to a model not yet loaded, Tool Adapter automatically loads this model to memory, and resolves the link to the linked model object. However, if the model does not exist in the workspace, Tool Adapter returns an empty object, which informs the tools of the model unavailability.

We note that Tool Adapter is a component that depends on the model representation used by each tool. Moreover, the on-demand loading functionality can be realized only when the model representation enables Tool Adapter to intercept navigation towards unloaded models. An example of such model representations is EMF.

On the other hand, for model representations lacking navigation interception support, we propose an alternative mechanism, which consists of allowing the tool to explicitly specify all models that it wants to load. In this case, Tool Adapter will disable the links from the loaded models to the unloaded models. We prefer disabling those links, rather than loading the linked models recursively, in order to avoid high memory consumption.

4.2 Delta extraction

In order to merge models, the developer executes the command merge (of WorkspaceManager), and specifies a set of models to be merged. WorkspaceManager begins model merging by extracting deltas from those models. To do so, it retrieves the Vbase version of those models from the repository. Then it compares the Vbase-

version models with the Vlocal-version models (in the workspace) for generating update commands.

Delta metamodel. We represent update commands in the form of models as well; i.e., we define the delta metamodel (cf. fig. 4), which define the update commands' structures. This metamodel contains the following metaclasses: Delta groups all update commands that affect a given model. UpdateCommand is a super type of all update commands. It specifies the *context element*, which the command involves, using its ID (cf. meta-attribute contextElemID). Create represents creation of an element. Delete represents deletion of an element. Move represents element moving to a target model, specified with a path (cf. meta-attribute targetModelPath). ModifyPrimitiveAtt, InsertRef, RemoveRef represent changes to an element's attributes. ModifyPrimitiveAtt represents changes to a primitive attribute, while InsertLink/ RemoveLink represent changes to a link attribute: link insertion/ removal. A link to be inserted/ removed is denoted by the element's ID and the model's path (cf. metaclass Link). InsertLink also supports changes to the ordered link attribute: it specifies the position to insert a link sequence (cf. meta-attribute positonAfter). This metamodel offers the following new features dedicated to large-scale models. (1) It expresses element moving, an essential action for reorganizing the partitioning of a modelbase. (2) It is based on the scalable addressing scheme, which enables us to link update commands to the involved elements in a large modelbase.

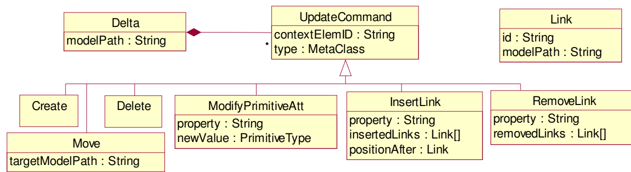

Figure 4. Delta metamodel

- **Comparison mechanism**. Our comparison mechanism is based on IDs (similar to [14] [9]). It consists of matching elements in two model versions, determining new elements, deleted elements, elements whose content is changed, and finally generating corresponding update commands. This mechanism offers the following new features. (1) It detects element moving. For two matching elements, if the mechanism determines that both elements belong to different models, it will generate a Move command. (2) It offers scalability. It can compare (two versions of) each model separately, so it reduces the amount of data to be loaded in memory at a time. The only case where multiple models need to be examined simultaneously is when elements are moved among those models.

4.3 Delta integration

In order to produce the merged version (Vmerged), WorkspaceManager retrieves the Vrep-version models from the repository, and applies the update commands to them. Then it commits the results back to the repository. During the command application, WorkspaceManager detects the lost update conflicts and ensures the consistency of the modelbase. We handle the link inconsistency problems caused by element moving and deletion as follows:

4.3.1 Handling element moving

We present the following solutions to the problems identified in section 2.3:

Solution for "updating moved elements". This problem is caused by the previous committer's moving commands that invalidate the current committer's the update commands. Let's suppose that one developer has moved element E from a source model Msrc to target model Mtar, and commits this change. Concurrently, another developer updates E. Then, he merges his update after the previous committer. Consequently, WorkspaceManager will detect that E is missing in Msrc, but it does not know whether E has been moved or deleted, and, in the case of moving, what the target model is.

For scalability, exhaustive search of the move target in the modelbase should be avoided. Therefore, we propose that WorkspaceManager maintains element moving history for each model, and stores it together with the model in the repository. This moving history is stored in a file called MoveLog, which is associated with each model. A MoveLog file contains a set of entries: *deletion entries*, and *move entries*. A deletion entry informs about an element's deletion, while a move entry informs about an element's moving from this model and also the move target model.

Therefore, to adapt an update command to missing element E in Msrc, WorkspaceManager examines Msrc's MoveLog file, then it can identify whether E has been deleted or moved. Let's suppose E has been moved, to apply a command involving E, WorkspaceManager will identify the move target model (Mtar). Consequently, it will adapt the command accordingly to the move target model: it will apply the command to Mtar.

Solution for "moving referenced elements". This problem occurs when the developer's move commands invalidate the existing links in the modelbase. Let's suppose that the developer wants to move element E from model Msrc to Mtar. The existing links to E in the entire modelbase need to be repaired, so that they point to Mtar.

For scalability, we aim to avoid exhaustive search for locating all links to E. Therefore, we propose that WorkspaceManager maintains index files and store those files together with the models in the repository. Each of those index files, called RefIndex, is associated to a model. It informs about all elements in other models that contain links to the elements in this model. WorkspaceManager updates RefIndex files each time a developer commits his updates: link insertion results in new entries in the RefIndex files, while link removal results in entry deletion in the RefIndex files.

Therefore, in order to repair links to a moved element E, first WorkspaceManager examines the RefIndex file associated with the model which is the source of moving (Msrc). Then, it locates all models owning links to E in the modelbase, without exhaustive search. Next, it retrieves those models from the repository, updates links in those models, and commits them together with the models that the developer explicitly commits.

4.3.2 Handling element deletion

We provide the following solution to the conflicts caused by element deletion, identified in Section 3.2.2.

Solution for "creating links to deleted elements". During command application, WorkspaceManager detects whether the elements referred to by a command is missing. Then, it will use the location mechanism (previously described) for determining whether the missing elements have been deleted or moved. In the case of deletion, the command will be marked as conflicting. The developer can choose to resolve this conflict either by applying our automated conflict resolution rules, or manually. We propose the following automated conflict resolution rules that the developer can choose: (a) the conflicting command is cancelled. (b) WorkspaceManager restores the deleted elements by copying them from Vbase to Vrep. For manual conflict resolution, WorkspaceManager enables the developer to manually edit the merged models before committing them to the repository, so that he can manually repair the model inconsistency.

Solution for "deleting referenced elements". Before WorkspaceManager applies a Delete command, it needs to verify that there is no link to the element to be deleted. In fact, WorkspaceManager applies Delete commands after RemoveLink commands, to remove links to the to-be-deleted elements first. However, since there can exist links originating from outside the model subset to be merged towards the to-be-deleted elements, which would become dangling links, WorkspaceManager needs to verify that no such link exists. To so do, it examines the RefIndex files, as previously described. If it detects potential dangling links, then it will mark the Delete command as conflicting. The developer can choose to resolve this conflict either by applying our automated conflict resolution rules, or manually. We propose the following automated conflict resolution rules that the developer can choose: (a) The conflicting Delete command is cancelled. (b) WorkspaceManager removes all the dangling links from the modelbase.

4.4 Example

Figure 3 illustrates a model merging example with ModelBus. In this example, the modelbase contains four UML class models (p1, p2, p3, p4). Each model can contain a set of UML classes, and the classes in different models can be interconnected. Two developers (Bob and Tom) retrieve these models (or subsets of them) to their workspace, and concurrently modify them. Then, Bob integrates his updates to the repository before Tom. When Tom integrates his updates, ModelBus will adapt the command "insert link C5→C4", according to the moving of C4 (cf. problem "updating moved elements"). Moreover, ModelBus also repair the link C6→C2 according to the moving of C2 (cf. problem "moving referenced elements").

This example also shows the detection of dangling link conflicts. ModelBus detects that the command of Tom "insert link C5→C3" is invalid, because C3 has been deleted by Bob (cf. problem "crating links to deleted elements"). The dangling link caused by command "delete C1" is also detected, since Bob has previously inserted the link C6→C1 (cf. problem "deleting referenced elements").

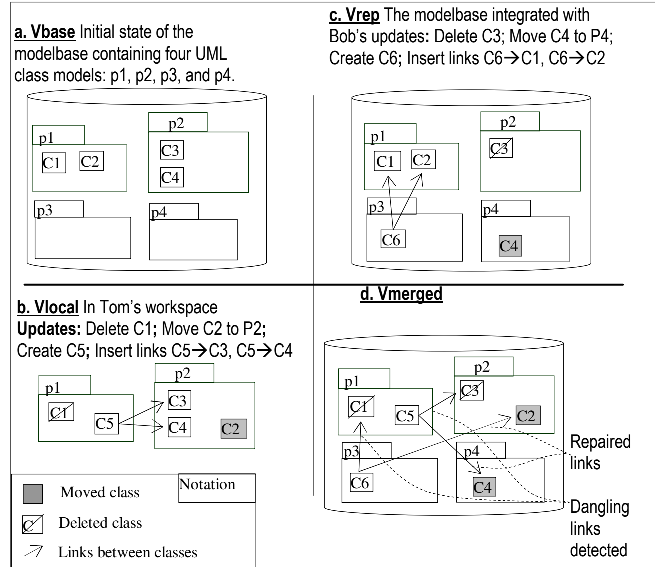

Figure 3. A model merging example

4.5 Implementation

The ModelBus implementation is available as an Eclipse project MDDi. This implementation is composed of the WorkspaceManager and Tool Adapter components. Tool Adapter is implemented in the form of an API, which existing CASE tools can call for loading and saving large-scale models. This component depends on the model representations used by the tools. The current implementation supports the EMF model representation, which is widely used by recent CASE tools. We estimate that implementing Tool Adapter for other model representations is not costly for tool programmers who want to adapt their tools to our approach. In fact, tool programmers can reuse existing XMI import/export modules for building Tool Adapter.

The WorkspaceManager implementation is independent from the CASE tools that are integrated with ModelBus: it can be executed as a separate program. It is composed of modules that manage the interaction with a CVS repository, and that perform the merging mechanism. It also offers user interfaces to developers (for retrieving, merging and committing models) through the Eclipse workbench.

ModelBus has been validated in the European project ModelWare. It offers the interoperability among multiple CASE tools which are provided by project partners (18 tools have been integrated so far).

5. Conclusion and future works

In this work, we have identified the key requirements on supporting collaborative work on large-scale models. Those requirements are not completely supported by OODB systems (since they lack tool interoperability), neither by existing model merging approaches (since they do not consider data partitioning). Then, we presented our experience in realizing ModelBus. Our solution can be adapted to current software projects, either by advanced developers or tool implementers, for solving difficulties in managing traceability links and inter-view links among models, which are currently scattered in different tools and different

developers' machines. We also note that our solution takes into account its usability in real software projects (interoperability with existing tools and genericity to all model types). Our approach does show the ability to limit the data amount to be processed at a time (for both model loading and model merging). For the future, we consider reporting performance measurements on different model merging scenarios. We aim to realize this task in the ModelPlex project (the follow-up project of ModelWare).

6. REFERENCES

[1] E. W. Adams, M. Honda, T.C. Miller, Object Management in a CASE Environment, Int'l Conf. on Software Engineering (ICSE), 1989.

[2] M. L. Barja et al., An Effective Deductive Object-Oriented Database Through Language Integration, Int'l Conf. on Very Large Data Bases, 1994.

[3] J. Bézivin S. Hammoudi, D. Lopes, F. Jouault, Applying MDA Approach for Web Service Platform, EDOC Conf., 2004.

[4] F. J. Budinsky, M.A. Finnie, J.M. Vlissides, P.S. Yu, Automatic Code Generation from Design Patterns, IBM Systems Journal, 1996.

[5] R. K. Dybvig, C. Bruggeman, D. Eby, Guardians in a Generation-Based Garbage Collector, ACM SIGPLAN Notices, 28(6), 1993.

[6] J. Grundy, J. Hosking, W.B. Mugridge, Inconsistency management for multiple-view software development environments, IEEE Trans. on Software Engineering 24(11), 1998.

[7] F. van Ham, Using Multilevel Call Matrices in Large Software Projects, IEEE Symp. on Information Visualization, 2003.

[8] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, The Generic Modeling Environment, Workshop on Intelligent Signal Processing, IEEE, 2001.

[9] A. Mehra, J. Grundy, J. Hosking, A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design, IEEE/ACM Conf. on Automated Software Engineering (ASE), 2005.

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

[11] J.P. Munson, P. Dewan, A Flexible Object Merging Framework, ACM Conf. on Computer Supported Cooperative Work (CSCW), 1994.

[12] C. Nentwich, L. Capra, W. Emmerich, A. Finkelstein, xlinkit: A Consistency Checking and Smart Link Generation Service, ACM Trans. on Internet Technology 2(2), 2002.

[13] T. Oda, M. Saeki, Generative Technique for Version Control Systems for Software Diagrams, IEEE Int'l Conf. on Software Maintenance (ICSM), 2005.

[14] D. Ohst, M Welle, U. Kelter, Difference Tools for Analysis and Design Documents, IEEE Int'l Conf. on Software Maintenance (ICSM), 2003.

[15] C. Oussalah, C. Urtado, Complex Object Versioning, Int'l Conf. on Advanced information Systems Engineering (CAiSE), LNCS, 1997.

[16] J. Rho, C. Wu, An Efficient Version Model of Software Diagrams, Asia-Pacific Software Engineering Conf. (APSEC), IEEE, 1998.

[17] J. Richardson, J. Green, Automating Traceability for Generated Software Artifacts, IEEE Conf. on Automated Software Engineering (ASE), 2004.

[18] M. Richters, M. Gogolla, Validating UML Models and OCL Constraints, UML Conf., 2000.

[19] P. Sriplakich, X. Blanc, M.-P. Gervais, Applying Model Fragment Copy-Restore to Build an Open and Distributed MDA Environment, MoDELS/UML Conf., 2006.

[20] P. Sriplakich, X. Blanc, M.-P. Gervais, Supporting Collaborative Development in an Open MDA Environment, IEEE Int'l Conf. on Software Maintenance (ICSM), 2006.

[21] P. Sriplakich, ModelBus – An Open and Distributed Environment for Model Driven Engineering, Ph.D. Thesis, University Pierre and Marie Curie, http://www-src.lip6.fr/homepages/Prawee.Sriplakich, September 2007.

[22] Y. Wang, D.J. Dewitt, J-Y. Cai, X-Diff: An Effective Change Detection Algorithm for XML Documents, IEEE Conf. on Data Engineering, 2003.

[23] D. L. Wells, J. A. Blakeley, C. W. Thompson, Architecture of an Open Object-Oriented Database Management System, IEEE Computer 25(10), 1992.

[24] Z. Xing, E. Stroulia, UMLDiff: an Algorithm for object-oriented design differencing, IEEE/ACM Conf. on Automated Software Engineering (ASE), 2005.

[25] Z. Xing, E. Stroulia, Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study, IEEE Int'l Conf. on Software Maintenance (ICSM), 2006.