Generative Technique of Version Control Systems for Software Diagrams

Takafumi Oda Motoshi Saeki
Dept. of Computer Science, Tokyo Institute of Technology
Ookayama 2-12-1-W8-83, Meguro-Ku, Tokyo 152-8552, Japan
saeki@se.cs.titech.ac.jp

Abstract

In iterative software development methodology, a version control system is used in order to record and manage modification histories of products such as source codes and models described in diagrams. However, conventional version control systems cannot manage the models in a logical unit because the systems mainly handle with source codes. In this paper, we propose a technique of version control based in a logical unit for models described in diagrams. Then we illustrate the feasibility of our approach with the implementation of version control functions on a meta-CASE tool that is able to generate a modeling tool in order to deal with various diagrams.

1 Introduction

In software development processes, various kinds of documents and source codes produced in the processes is frequently changed by various reasons, e.g. customer's requirements changes, even during its development. Developers should have various versions of a product and manage them in their project. In this situation, the techniques for version control are significant to support their tasks by using computerized tools. In addition, in modern software development, we frequently adopt iterative and incremental development styles such as Unified Process [2] and XP [3], and version control is mandatory for incremental and iterative development styles. We have excellent version control techniques and computerized supporting tools for source codes such as RCS [4], CVS [5] and Subversion [6]. These tools store the current version of a product and the differences between the adjacent versions in a repository, so that it can recover all of the older versions by applying the stored differences to the current one (backward difference). However, they are for text documents and adopt line based management, i.e. the granularity of version control is a "line" and the difference is generated line by line. Since we use diagram documents such as class diagrams in modeling stages of development projects, we should manage the changes on the diagrams, not in the granularity of a line, but of a logical unit component, e.g. "Class", "Association", "Attribute" etc. in the case of Class Diagram. The targets of version control should be logical components and they depend on development methods. For example, the targets are "Class", "Attribute", "Association", etc. in the case of Class Diagram, while they are "State", "Transition" etc. in State Diagram. That is to say, we should manage version records in the level of manipulating components of a class diagram, e.g. creation of a class and deletion of an attribute in a class, etc.

To model a complicated software system, we represent its model with various diagrams from separated multiple viewpoints, for example a class diagram for structural views and state diagrams for behavioral view. It means that our version control system should handle with various logical components that are different according to the used diagrams. We should have an individual version control system for each diagram and the development of these various version control systems consumes much wasteful labor.

Some of CASE tools for diagrams such as Argo UML [7] can transform a diagram into a XML document by using XMI technology [9], and after the transformation, we can apply to the transformed XML document an existing version control tool such as RCS and CVS. In fact, some of them such as Eclipse UML [11], Jude[10] and Poseidon[12] have interfaces to CVS. However, this solution cannot achieve an integrated and seamless support for developing a diagram and for its version control, and the developers should use two separated tools; a CASE tool and a version control one. And as far as RCS or CVS is used, the management of versions is based on the granularity of a line, not a logical component of a diagram. Some tools such as Rational Rose [13] and Konesa [14] include the function of "differencer" [15, 16], which shows the differences between the older version of the diagram and the current one, e.g. by highlighting different graphical components or by depicting in tree form the changed classes appearing in a class diagram. "Differencer" holds all versions of a product thoroughly, not the differences between them. Thus it has a shortcoming that large amount of spaces in the repository is wasted. Furthermore, we have to construct each differencer for each type of diagram, e.g. Differencer for Class Dia-

Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)
1063-6773/05 $20.00 © 2005 **IEEE**

gram, Differencer for Sequence Diagram, etc., and much effort in building development environment is necessary. In [18], a technique of version control for diagrams was discussed. However, it regarded a diagram as a graph consisting of nodes and edges and it did not consider logical properties specific to the diagrams. The techniques to calculate differences on XML [20] and on complex objects [17] were studied. However, they did not discuss the flexible variation of the logical components that should be targets of version control.

In this paper, the version control technique based on logical components of diagrams and a kind of generator of version control system is proposed to solve the above problems. The essential point of our work is generating a CASE tool together with the functions of version control from a meta model representing a diagram. That is to say, we have developed a kind of meta-CASE tool generating the tools having version control mechanisms. The rest of the paper is organized as follows. Section 2 presents the outline of the proposed system and introduces our meta-CASE tool, a kind of CAME (Computer-Aide Method Engineering tool) [1]. A meta modeling technique to represent diagrams is also shown in the section. Section 3 describes the operations of modifying a diagram in order to represent differences between versions. In sections 4 and 5, we present the functions of version control and the illustration of the generated diagram editors having version control mechanism, respectively. Section 6 is the concluding remark and discusses the future work.

2 Generating Version Control Systems

2.1 The Overview of Our Approach

In our version control system, we adopt a technique to store differences between two versions in a repository like RCS and CVS, etc. so that we can recover the older versions that were previously produced. In this approach, the state of the artifact at a certain time is considered as a baseline, and the version control system stores the difference between this baseline and each version to the repository. It has an advantage point where products are efficiently manageable.

When adopting an approach of storing differences, the techniques to define the differences between versions and to acquire them are necessary. To extract a difference efficiently, we focus on the developer's activities of editing a diagram by using a diagram editor. In other words, we generate the element of the difference from an execution of an editor operation such as "create" and "delete" a component. Suppose that a developer creates a new class in a class diagram by using Class Diagram editor. The execution of the editor operation "create a class" results in its addition to the difference data. The meta model of the diagram provides the information on what editor operations we should focus on, and we can specify operations such as "create", "delete" and "update" a logical component appearing in the meta model. The sequence of the editing operations that a developer is performing is acquired in real-time during his editing activity using the editor. The acquired operation sequence can be considered as the difference between versions, and is stored in the repository. Our meta-CASE tool, which generates a diagram editor from the meta model description, should automatically embed the functions of acquiring performed editing operations in real-time and of transforming them to difference data, when it generates the editor.

Figure 1 depicts the overview of our system. As shown in the figure, we have two types of engineers; one is called method engineer who is the expert of specifying meta models and another is a software engineer who constructs models of the software systems to be developed. The method engineer uses a method editor to manipulate the meta model. The method editor is a kind of diagram editor because Class Diagram (precisely, MOF) is adopted to describe a meta model, and it allows the method engineer to easily edit meta models.

The details of the meta model will be explained in the next sub section. Our meta-CASE tool automatically generates from the meta model, 1) an editor (a modeling tool) for supporting inputting and editing products, such as the editor of Class Diagram, and 2) the schema of a repository. That is to say, the meta model serves as the schema definition for the repository to which the developed products are stored. Software engineers may develop a model of a software system by using the tools generated from the meta model. The functions of version control, whose access interfaces are similar to CVS, are embedded into the generated diagram editors.

A software engineer, i.e., a user of the generated editor develops the first version of a model as a baseline, and imports it to the repository. He can check out any versions of the model that are stored in the repository, and edit them by using the editor. The editor gets the operation sequences on the model in real-time by monitoring the editor commands that he used. And he can check in the current version of the product by storing operation sequences as the difference to the repository whenever he wants to do.

2.2 Meta Models and Meta-CASE

The example of the meta model of the simplified version of use case diagrams is shown in a left side window of Figure 2. As shown in the figure, the meta model "Use Case Diagram" has the concepts "Actor", "Association", "UseCase" and "System" and all of them are defined as classes on a meta model. The concept "ModelElement" is their super class and has the attribute "name", and the instances of UseCase, Actor, Association and System can have their names. These concepts (called method concepts) have associations (called method association) representing logical relationships among them. For instance, the concept "UseCase" is associated with "Association", so use cases can be

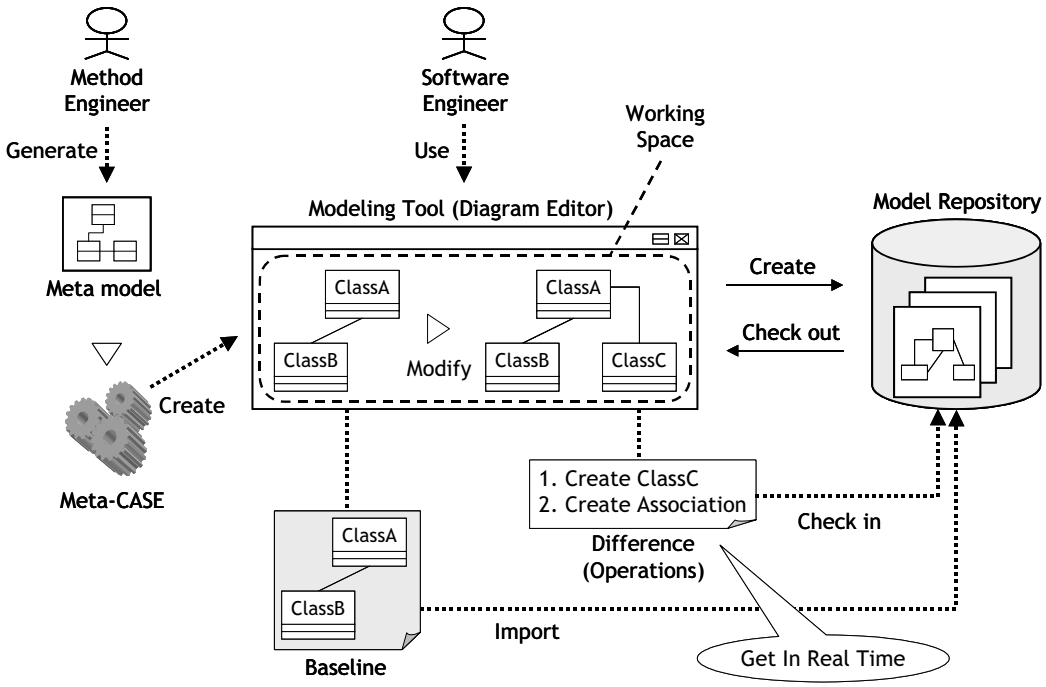


Figure 1. Overview of Our System

connected with each other through the instances of "Association". This is for representing the relationships such as "uses" and "extends" between use cases. We simply call both method concepts and method association method elements. The method elements can be the logical unit of version control.

In addition, we should consider constraints on the products. For example, we cannot have different use cases having the same name in a use case diagram. We can specify this constraint to keep consistency of the products, i.e. class diagrams on its meta model, by using OCL (Object Constraint Language). On account of space, we do not explain more details of our meta modeling technique, and the readers can find them in [1].

Our meta-CASE is only for generating diagram editors which deal with a product conceptually as a graph consisting of nodes and edges. Thus we should provide the information on which the method concepts in a meta model can be represented with nodes or edges of the graph. The method engineer provides two types of information: one is the correspondence of method concepts to the elements of graphs, i.e. nodes, edges, texts within the nodes and texts on the edges, and another is notational information of the nodes and edges. Suppose that he tries to generate a use case diagram editor from Use Case Diagram. The concept UseCase, Actor and System in the Use Case Diagram conceptually corresponds to nodes in a graph, while Association does to edges. He provides this information as stereotypes on the method concepts. Figure 2 included the information for the meta-CASE as well as the meta model of use case diagrams. The readers can find the stereotypes "<<Entity>>" and "<<Relationship>>" attached to the classes in the meta model. The former stereotype stands for the correspondence to a node and the latter to an edge. For example, an occurrence of a use case in a use case diagram corresponds to a node from the viewpoint of the graph, while an association between use cases to an edge. Note that a generated editor automatically includes commands for creating and deleting the method concepts corresponding to the nodes or the edges.

In addition, method engineer should specify which figures, say a rectangle, a circle, an oval and a dashed arrow, are used for expressing method elements on the editor screen. We call this information *notation model*, while the model consisting of method concepts and associations for expressing its logical structure is called *logical model*. Basic graphical figures such as figures used in UML diagrams are built-in and their drawing programs are embedded as Java classes into our meta-CASE. A method engineer selects the figures out of these pre-defined built-in figures for the <<Entity>> components and <<Relationship>> ones, by clicking a menu item, as shown in the left window Meta-CASE of Figure 2. In the example of this figure, the method engineer specifies a graphical figure for Actor as "ActorShape" which is pre-defined and built-in shape in the system. The meta-CASE produces a diagram editor by embedding the above information and Java classes into a modeling tool framework, as shown in Figure 3.

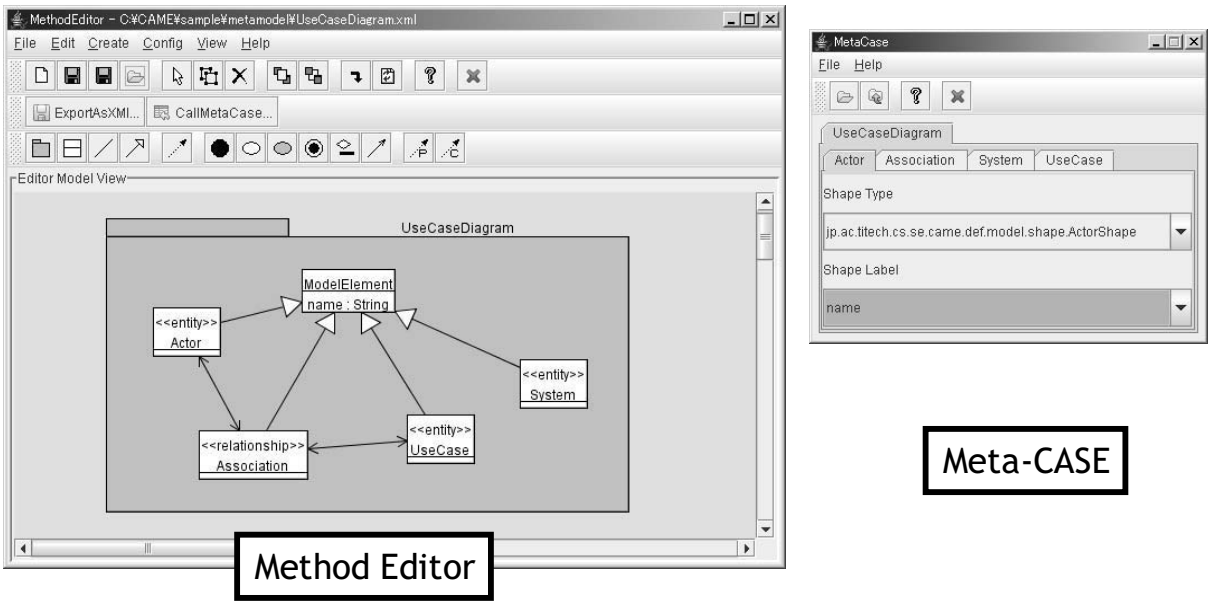


Figure 2. An Example of A Meta Model and A Meta-CASE

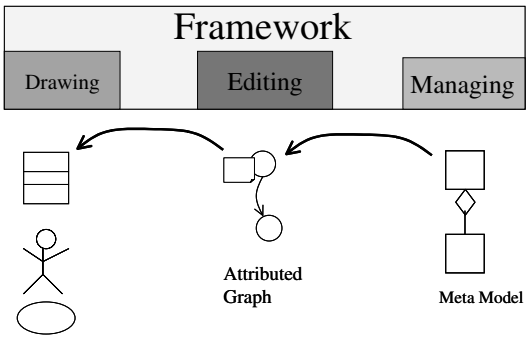


Figure 3. Generating Diagram Editors

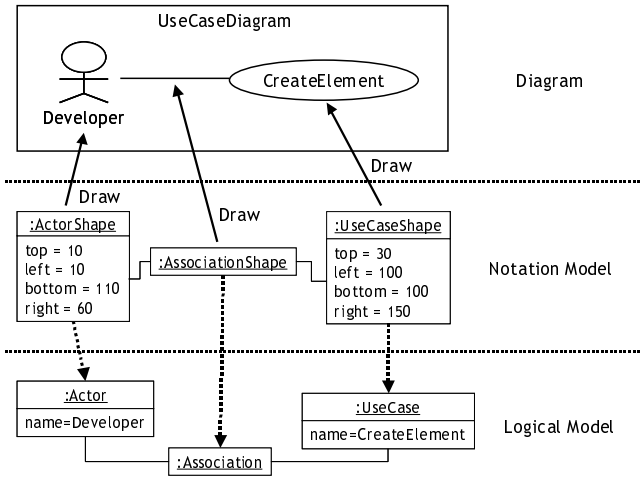


Figure 4. An Instance of Use Case Diagram

Figure 4 illustrates the instance of a use case diagram and shows how to store it following its meta model. As mentioned above we have a logical model and a notation model. The logical model has method concepts "Use Case" and "Actor", both of which have the information on their names, e.g. "CreateElement" and "Developer". They are connected to the elements of the notation model having the information on their locations and the graphical shapes on the screen, as depicted with dotted arrows in the figure. All model elements included in logical models and notation ones that can be the targets of version control hold unique identifiers called UUID [19], in order to keep their identities in the repository.

3 Operations on Model Elements

In this section, we discuss operations on the models to represent the differences between versions. The operations can be classified into the following three categories:

- Operations on a logical model
  They are for inputting and editing method elements on the logical part of the meta model, e.g. creating and deleting method concepts such as Class, State and Association etc.

- Operations on layout data of graphical elements
  They are for changing layout of graphical elements on the display screen, such as moving and resizing a rectangle denoting the class. These operations do not have any semantic influence on the model.

- Operations on a notation model
  They are for manipulating the elements of the notation model such as creating a node and deleting an edge etc.

These operations include the UUIDs as parameters so that their target instances can be uniquely identified in the repository. In addition, the instances of these operations are labeled with unique UUIDs so that the version control system can identify them.

Tables 1 and 2 show the operations on the layout data of the graphical elements and on the notation model respectively. The control point in table 1 is a point where a user can change the size of a graphical element by his dragging with a mouse.

Note that the delete operations such as DeleteEdgePickPoint and DeleteNode have the information on the deleted elements as their parameters. For example, DeleteNode operation includes location data x and y which denote X-Y coordinates of the element on the display screen. This information is necessary to implement Redo/Undo functions in the generated modeling tools. If a user deletes an element and the corresponding delete operation does not have the X-Y coordinates of the deleted element, the tool cannot recover and put the deleted element on the screen when he invokes Undo. Redo/Undo functions are implemented by using these operations, shown in the tables, in the modeling tools.

The operations on layout data and on a notation model are common to any diagrams, while the operations on a logical model are dependent on diagrams because we should have the operations specific to method elements, e.g. creating a *Class* for Class Diagram and creating a *State* for State Diagram. We extract these operations from a meta model focusing on its method concepts. Furthermore, we have several additional operations to manipulate the relationships between a logical model and a notation one, and to edit the texts of attributes of the method concepts. For example, the following two operations are to create and delete a connection between a logical model eUUID and a notation model whose type is sType.

ConnectShapeModel(UUID, eUUID, sType)
DisconnectShapeModel(UUID, eUUID, sType)

The following three operations are for updating, creating and deleting the texts of the attributes attached to an element eUUID of the model.

UpdateAttribute(UUID, eUUID, sUUID, attrName, preValue, newValue)
Create*Element*(UUID, eUUID, eType, *attr1*, *attr2*, ...)
Delete*Element*(UUID, eUUID, eType, *attr1*, *attr2*, ...)

Table 1. Operations on Layout Data

Operations	Definitions
Moving a control point	MovePickPoint(UUID, sUUID, pID, dx, dy) Moving a control point labeled with pID in a notation model sUUID (dx, dy).
Moving a graphical element	MoveShape(UUID, sUUID, dx, dy) Moving a notation model labeled with sUUID (dx, dy).
Inserting a vertex in a polygonal line	CreateEdgePickPoint(UUID, eUUID, pID, x, y) Creating a control point pID at (x, y) in the edge eUUID.
Deleting a vertex in a polygonal line	DeleteEdgePickPoint(UUID, eUUID, pID, x, y) Deleting a control point pID in the edge eUUID.

Any editing activity on a diagram through a modeling tool can be defined as the combination of these operations. In this sense, these operations listed above are primitive and atomic. Suppose that a software engineer creates a new class and add it to a class diagram. This activity comprises 1) creating a class in a logical model of Class Diagram, 2) creating a node (a rectangle box) on a display screen, and 3) connecting the class in the logical model to the node in the notation model. Thus we can have the sequence of the operations as follows.

1. CreateClass(UUID1, ClassUUID, anonymous)
2. CreateNode(UUID2, RectangleUUID, ClassShape, 30, 20)
3. ConnectShapeModel(UUID3, RectangleUUID, ClassUUID)

In the above description, UUID1, UUID2 and UUID3 stand for the unique identifiers denoting the three operations. The first operation creates a class in the logical model level and labels it with ClassUUID, while a rectangle labeled with RectangleUUID is created and put at the location (30,20) of the screen in the second operation. ClassShape is the type of graphical element, i.e., a rectangle in this case, which is pre-defined in the meta-CASE. The last operation is for connecting the two components that have been created by the previous operations.

For each execution of an editing command of a modeling tool, a combination of the corresponding operations is automatically generated as an element of the difference. In addition, to detect the conflicts on merging branched ver-

Table 2. Operations on a Notation Model

Operations	Definitions
Creating a node	CreateNode(UUID, nUUID, Type, x, y) Creating a node of type "Type" labeled with nUUID, at the position (x, y) on the screen.
deleting a node	DeleteNode(UUID, nUUID, Type, x, y) Deleting a node labeled with nUUID.
Creating an edge	CreateEdge(UUID, eUUID, Type, sUUID, tUUID) Creating an edge of type "Type" labeled with eUUID. The created edge is connected to the node sUUID as its source and to tUUID as a destination.
Deleting an edge	DeleteEdge(UUID, eUUID, type, sUUID, tUUID) Deleting an edge labeled with eUUID.

sions, the operations have pre conditions that should be true when they are applied, as mentioned later in section 4.4.

4 Version Control Functions

4.1 Overview

By using our generated version control system, a software engineer has working spaces at his local site, and performs import, check-out and check-in operations between his working space and a repository. He uses an import operation to get a working space and to import a product from the repository as a baseline. When the engineer checks out from the repository the version n of a product, a working space for modifying it is allocated at his local site and it is loaded into the working space. The engineer uses the modeling tool to modify the version n, and after completing the modification, he stores it as version n+1 into the repository (check-in).

A method engineer starts a method editor and develops a meta model of a diagram that software engineers will use. And then he generates a modeling tool for the diagrams specified with the meta model. The generated tool has menu commands for version control. The basic commands for version control can be listed up in the following;
import : for generating an empty working space and then loading a product as a baseline.
check-out : for loading an product from a repository to the current working space.
check-in : for saving the product in the current working space as a new version back to the repository.
How to use these commands is illustrated in the subsequent sections.

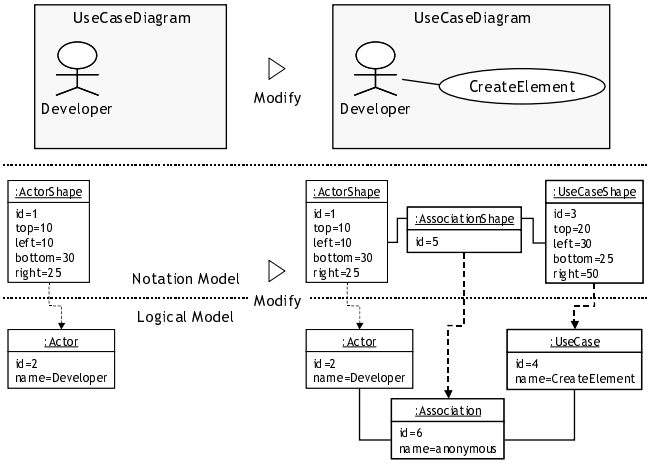


Figure 5. Acquiring Differences

4.2 Acquiring Differences and Checking In

Figure 5 illustrates how to acquire the differences between two use case diagrams. The figure includes three layers; the upper one depicts instances of use case diagrams, the middle and the bottom ones show their notation model and logical one respectively. Note that for easiness to read the figure, we simplify the identifier numbers of the elements (id) instead of UUID.

A software engineer, a user of the modeling tool for use case diagrams, develops a use case diagram as shown in the left side of the upper layer of the figure and sets it as a baseline. The use case diagram is stored as version 1 into the repository by this activity. Subsequently, he performs the following modifications and gets a new use case diagram as shown in the right side of the upper layer of Figure 5.

1. Creating a new use case. The default value is assigned to the attribute value "name", i.e. the name of the use case, by the editor.
2. Changing the name of the new use case to CreateElement.
3. Creating an association between the actor Developer and the new use case CreateElement.

The above sequence of the modifications causes the change of the attributed graph internally representing the use case diagram and the version control system gets the operation sequences shown in Figure 6. For example, the modification activity of creating a new use case in Compound(op1) of the figure comprises three operations: CreateNode in a notation model level, CreateUseCase in a logical

```
Compound(op1) {
  CreateNode(op2, 3, UseCaseShape, 30, 20)
  CreateUseCase(op3, 4, UseCase, anonymous)
  ConnectShapeModel(op4, 3, 4)
}
UpdateAttribute(op5, 4, 3, name, CreateElement, anonymous)
Compound(op6) {
  CreateEdge(op7, 5, AssociationShape, 1, 3)
  CreateAssociation(op8, 6, Association, anonymous)
  ConnectShapeModel(op9, 5, 6)
  AddAssocToUseCase(op10, 4, Association, 6)
  SetUsecaseToAssociation(op11, 6, UseCase, 4)
  AddAssocToActor(op12, 2, Association, 6)
  SetUsecaseToAssociation(op13, 6, UseCase, 2)
}
```

Figure 6. An Acquired Operation Sequence

model level and ConnectsShapeModel for relating the notation model to the logical model. The word "anonymous" appearing as the fourth parameters of CreateUseCase stands for a default name of the created use case. The second operation UpdateAttribute is for the activity of replacing it with CreateElement. Compound(id6) is for establishing an association between the actor and the use case.

The user can check in the acquired operation sequence as a difference to the repository whenever he wants to do. The version number is automatically incremented and it is labeled with version 2 if he does not explicitly specify any version number. If he specifies the number, it is stored as one of the branching versions of version 1.

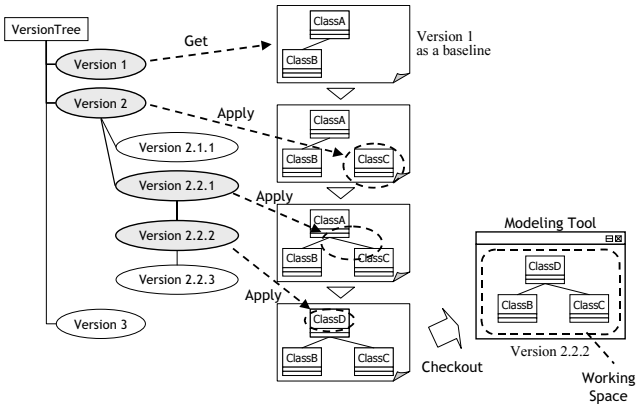
4.3 Checking Out

The engineer can check out a specific version to his working space, by applying the stored differences to the version 1 that is a baseline. Figure 7 shows the example of checking out the product of version 2.2.2, and its left part depicts a version tree.

To check out the version 2.2.2, first of all the engineer should get the version 1, that is a root of the version tree, as a baseline by using "import" command. The path in the tree from the version 1 to any version being checked out can be uniquely identified because this version tree is a *real* tree. To get the version 2.2.2, the version control system traces the path to it in the tree and applies to the root version all differences that the path holds one by one (i.e., the differences from the root version 1 to the version 2.2.2). By these successive applications, the engineer can get and manipulate the version 2.2.2 in his current working space.

4.4 Merging Branching Versions

Similar to CVS, our version control system and repository can have branched versions. To deal with branched versions, the function of merging different versions that are branched into a new version is necessary. This can be done by applying to the baseline the differences that the branched



Scenario

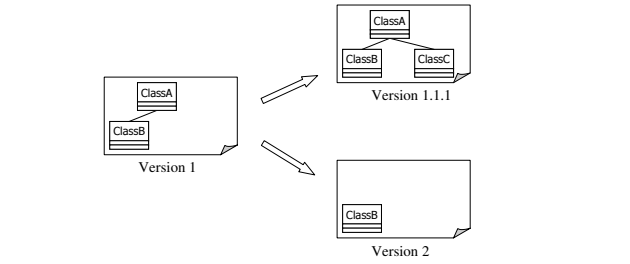
Version 1 : Class A + Class B + an association between A and B

Modifications:
from Version 1 to Version 2 : Creating a class C
from Version 2 to Version 2.2.1 : Creating an association between A and C
from Version 2.2.1 to Version 2.2.2 : Changing the class name of A to D

Figure 7. Checking Out

paths hold and it is the same technique as a check-out function. On merging, an engineer has two versions: one is for playing a role of a baseline and another is the merged version. We call the former base version and the latter merge version. The merge version is merged into the base version. The procedure how to merge the version A to the version B is outlined as follows. Our version control system goes up to the root of the version tree and looks for a version that is a common ancestor to the versions A and B. After checking out the version B as a base version, it applies to the version B the differences that are held on the path from the common ancestor to the version A (merge version). The only point different from the check-out function is to handle with the occurrences of conflicts among the branched versions, i.e. the base version and the merge one.

There may occur conflicts when applying the differences to a base version. See a scenario of the modification shown in Figure 8, and suppose that the version 1 had classes A and B and that the class A was deleted at the version 2 from it. In the version 1.1.1, which is one of the branched versions from version 1, a new class C is created and associated with the class A. The engineer selects the version 2 and the version 1.1.1 as a base version and a merge one respectively, i.e. he will merge the version 1.1.1 to the version 2. If the system tries to apply to the version 2 the difference from version 1 to 1.1.1, it fails because the class A does not appear and it is impossible to establish the association between A and C in the version 2. However, since it is possible to create a class C, its operation is applied and as a result the engineer gets the version 2 having the class C. In this case, the system records the operations that are failed to be applied, e.g. CreateEdge between A and C,



Scenario

Version 1 : Class A + Class B + Association between A and B

Modifications:

from version 1 to version 2 : Deleting a class A and an association between A and B

from version 1 to version 1.1.1 : Creating a class C and an association between A and C

Figure 8. A Modification Scenario Example

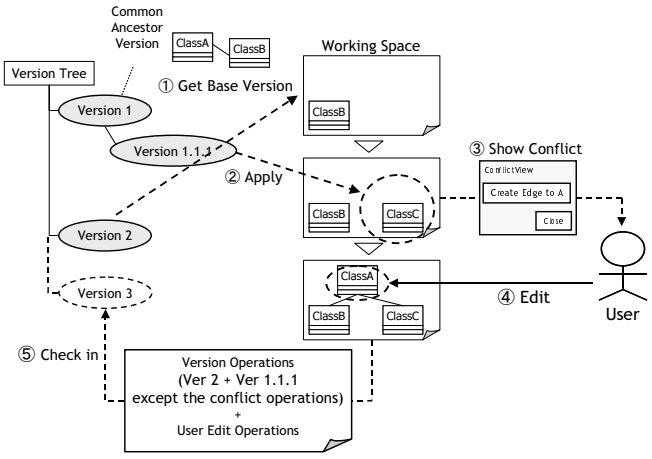


Figure 9. Conflicts in Merging

and continuously performs the operations that are applicable. In this example, only CreateClass C is performed. To detect conflicts, we attach to each operation a pre condition that should hold before its application. For example, the condition "the elements specified with sUUID and tUUID shall exist" is attached as a pre condition to the operation CreateEdge(UUID, eUUID, type, sUUID, tUUID), which connects sUUID to tUUID. If the condition is evaluated to be false, its application is skipped and its failure is record as illustrated above. The recoded failures of applying operations are shown to the engineer if any, and he can edit by manual the diagram referring to the failure record, as shown in the steps 3 and 4 in Figure 9. After completing the merge activity, he checks in the diagram as the next version to or one of the branched versions from base version 2.

Note that the direction of merging may have the influence in the result. For example, although merging the version A to B has no conflicts, merging B to A may have a conflict. Return back to the above example and consider

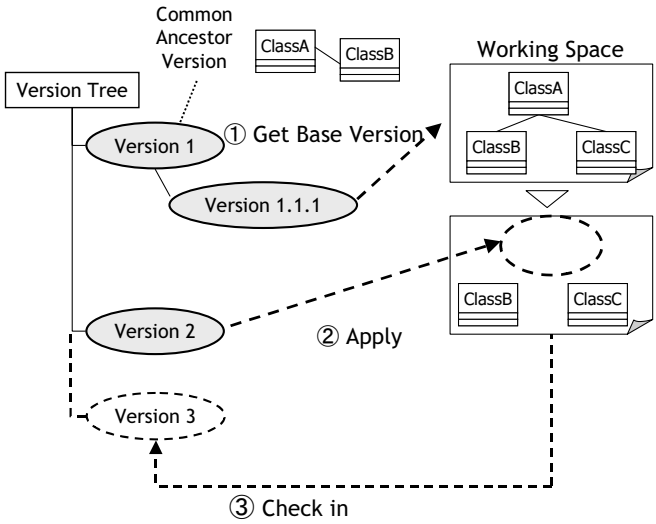


Figure 10. No Conflicts in Merging

that the engineer will merge the version 2 to the version 1.1.1, whose merge direction is reversed with Figure 9. As shown in Figure 10, the version 1.1.1 having the classes A and C is checked out as a base version, and during a merging process of the version 2, the class A is to be deleted by applying the differences from the version 1 to 2. This deletion is successful because the class A exists in the base version. The associations between A and B and between A and C are automatically deleted together with the deletion of the class A.

5 Implementing a Version Control in Meta-CASE

In order to assess the feasibility of our proposed approach, the version control functions have been implemented on the meta-CASE called CAME [1]. The tool comprises the following modules.

- **Modeling Tool Framework**
  It provides general functions of modeling tools. By filling with suitable software modules the hot spots of the framework, it is made a modeling tool specified by a meta model.

- **Method Editor**
  It is a graphic editor to input and edit meta models.

- **MetaCASE**
  By providing a meta model that is developed with the method editor, it automatically generates a modeling tool for inputting and editing diagrams following the meta model.

The modeling tool framework and meta-CASE are extended so that they can generate modeling tools having version control mechanism. As a result, we can have modeling tools with unified user interface of version control functions for various diagrams such as ER diagrams and UML ones.

Our repository has three types of file; 1) a file for a version tree, 2) files for baseline versions and 3) files for differences. We have a meta model for version trees and a meta model for the operations mentioned in section 3 as their abstract syntax, in addition to the meta model of a software diagrams such as Figure 2, corresponding to the above three types of file. In order to store data into these files, we use XML representation which is produced by Java library java.beans.XMLEncoder/XMLDecorder. This library can transform java objects into XML documents directly following these meta models.

By using the function of our meta-CASE tool, we actually generated two diagram editors: ER diagram editor and Use Case diagram one. By using these tools, our software engineers could construct these diagrams of a web application of a simple seat reservation system and handle with their versions, without any troubles.

The one observed weak point is time efficiency of a check out operation. Since we adopted the technique to hold the forward differences where the newest version is generated by applying all differences from the baseline version, it took longer time to check out the newest one. It took a couple of seconds to check out the version 2 from the version 1, where their difference included 160 operations.

The screen of a use case diagram editor are shown in Figure 11. This screen is the product of the version 4 whose difference from the version 3 is adding a new use case "Login". The window of "Difference of Selected Version", which is located in the right side of Version Tree Viewer, shows the sequence of the operations of this addition. It is possible for users to see differences not only by expressing an operation sequence in textual format, but also in intuitive and graphical display style, e.g. the tool can display the differences by highlighting or changing colors of the modified elements on the screen.

6 Conclusion and Future Work

In this paper, the version control technique based on logical components was proposed for models described by diagrams such as ER diagrams and UML ones. We also implemented a kind of Meta-CASE to generate modeling tools having unified version control functions based our proposed technique.

We can list up future works as follows:

- **Improving time efficiency**
  As discussed in section 5, time performance of a check out operation should be improved in order to make our tool more practical. In our tool, users can make any version as a baseline version and store it completely to the repository, by using import command. However, for the users, it may be a troublesome to consider which are major versions to be stored completely and re-define them whenever a new version is generated. Like CVS, we take the newest version as a baseline and hold the differences from it to the older versions, i.e., we will adopt the approach based on backward difference. In backward difference approach, all of the leaves in a version tree should be baseline versions and merge processing becomes more complicated. In our prototype implementation, we had taken a simpler approach, i.e. forward difference one.

- **Compliant to XMI**
  In this paper, although our tool can import and export the models in XMI-compliant format, we defined the representation of differences by using operations that have original syntactical structures. To standardize our tool, we should adopt more general representation technique such as XMI. We can use XMI.update operations to represent differences. They are used for informing the differences of XMI-compliant documents when the documents are exchanged. We have three operations; XMI.add for adding an element to the older document, XMI.delete for deleting an element, and XMI.replace for replacing an element with a new element. Adopting XMI.update operations is one of future works.

- **Applying to other CASE tools**
  We adopted the records of actually used editing operations to calculate the differences between two consecutive diagrams, so called operation based versioning. This approach leads to less applicability to the other existing CASE tools, and the tools having our version control technique are limited to what our meta-CASE generates. The implementation of *meta differencer*, which generates from a meta model a program to calculate the differences between two diagrams, is one of the interesting topic.

- **Support for multi-users**
  In this paper, we explained our approach in single-user and local-site situation. However, the support for distributed, concurrent and collaborative tasks by multi-users is significant. Our tool has the function to support version branching, merging and conflict processing in addition to highlighting the difference on the editor screen. For example, a user can be aware of the difference of his version to another version that the other user is producing by the highlighting function. Our tool has the function to support version branching, merging and conflict processing. Since multi-users can use our tool under uncontrolled situations, its function seems to be the first step toward multi-user support. To elaborate the supports for multi-users, we have to investigate real collaborative version control tasks on

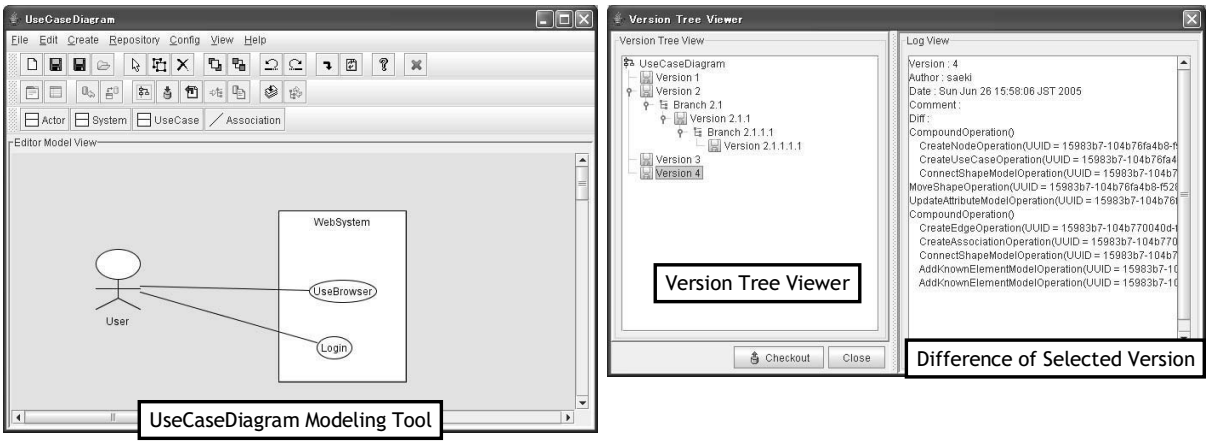


Figure 11. Display Screen of the System

software diagrams, which may be different from programs.

- **Optimizing an operation sequence in a difference** An extracted operation sequence can include useless operations for recovering and checking out the newer version. Suppose that a user created a class in a class diagram and then immediately deleted it. In our current approach, these two activities are held in the difference data. However they should be excluded when registering the difference to the repository, because they had no effects on the diagram before and after performing them.

References

[1] M. Saeki. Toward Automated Method Engineering: Supporting Method Assembly in CAME. In *Engineering Methods to Support Information Systems Evolution - EMSISEf03, OOIS'03* http://cui.unige.ch/db-research/EMSISE03/
2003.

[2] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.

[3] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, 1999.

[4] Revision Control System, http://www.cs.purdue.edu/homes/trinkle/RCS/.

[5] Concurrent Versions System, http://www.cvshome.org/.

[6] Subversion, http://subversion.tigiris.org/.

[7] ArgoUML, http://argouml.tigris.org/.

[8] UML Specification, http://www.omg.org/.

[9] XML Metadata Interchange, http://www.omg.org/.

[10] Jude, http://objectclub.esm.co.jp/Jude/.

[11] EclipseUML, http://www.omondo.com/.

[12] Poseidon, http://www.gentleware.com/.

[13] Rational Rose, http://www-6.ibm.com/jp/software/rational/.

[14] Konesa, http://www.canyonblue.com/.

[15] D. Ohst, M. Welle, and U. Kelter, "Difference Tools for Analysis and Design Documents," Proceedings of the IEEE International Conference on Software Maintenance, pp. 13-22, 2003.

[16] D. Ohst, M. Welle, and U. Kelter, "Differences between versions of UML diagrams," ESEC/FSE2003, pp. 227-236, 2003.

[17] C. Oussalah and C.Urtado, "Complex Object Versioning Source," Lecture Notes In Computer Science, Vol. 1250 (CAiSE97), pp. 259 - 272, 1997.

[18] J. Rho, and C. Wu, "An Efficient Version Model of Software Diagrams," Proceedings of the Fifth Asia Pacific Software Engineering Conference, pp. 236-243, 1998.

[19] Universally Unique Identifier, http://java.sun.com/j2se/1.5.0/ja/docs/ja/api/java/util/UUID.html.

[20] Y. Wang, D. DeWitt, and J. Cai, "X-Diff: An Effective Change Detection Algorithm for XML Documents," Proceedings of the 19th International Conference on Data Engineering, pp. 519-530, 2003.