

論文 / 著書情報
Article / Book Information

Title	Configuration Management in a Method Engineering Context
Author	Motoshi Saeki
Journal/Book name	Lecture Note in Computer Science, Vol. 4001, No. , pp. 383-398
発行日 / Issue date	2006, 6
DOI	http://dx.doi.org/10.1007/11767138_26
権利情報 / Copyright	The original publication is available at www.springerlink.com .
Note	このファイルは著者（最終）版です。 This file is author (final) version.

Configuration Management in a Method Engineering Context

Motoshi Saeki

Dept. of Computer Science, Tokyo Institute of Technology
Ookayama 2-12-1, Meguro-ku, Tokyo 152, Japan
Tel. +81-3-5734-2192 Fax +81-3-5734-2917 E-mail saeki@se.cs.titech.ac.jp

Abstract. Method Engineering is the discipline for exploring techniques to build project-specific methods for information system development and Computer Aided Method Engineering (CAME) is a kind of computerized tool for supporting the processes to build them. In this method engineering environments, version control and change management for both model descriptions and method descriptions should be seamlessly combined. In addition, when the method being used is changed during a project, we should check whether the current version of a model is still consistent to the newer version of the adopted method. This paper proposes a technique to solve the issues on version control and change management in method engineering processes.

1 Introduction

Development methods for information systems (simply, methods) and their supporting tools are one of the most significant key factors to get a great success of development projects. To enhance the effect of methods used in a development project, we need to adapt the methods or build the new ones so that they can fit to the project. Method Engineering is the discipline for exploring techniques to build project-specific methods for information system development, called situational methods. Computer Aided Method Engineering (CAME) is a kind of computerized tool for supporting the processes to build them [7].

Although we can have a powerful situational method, another difficulty originating from frequent changes of a product still remains. An artifact is frequently changed by various reasons, e.g. customer's requirements changes, even during its development. Developers should have various versions of an artifact and manage them in their project. In this situation, the techniques for version control and change management, i.e. for configuration management, are significant to support their tasks by using computerized tools. In [10], we have developed a version control system for model descriptions that are represented in diagrammatic form such as UML diagrams. Unlike the version control systems for text documents, our system can handle with logical components, e.g. "Class" and "Attribute" in a class diagram, as units of version control.

In method engineering environments, as well as changes of a model description, the description of the adopted methods may be changed. Therefore the support for version control and change management of methods themselves is

necessary. In [11], the changes of methods were classified into a set of patterns, but it did not mention any support for the version control of methods themselves.

In change management, there exist the dependencies among the components of an artifact, and the change of a component may be propagated to the other components having the dependencies to it, in order to keep consistency on them. This kind of change management should be done 1) on model descriptions (simply, product), 2) on method descriptions (simply methods or method fragments) and 3) on both of them. The third case is as follows; when the adopted method is changed, the change is propagated to the model description that was developed with the older version of the method. Model management systems such as Coral [3] and UML repository systems [5, 8, 12] are only for meta models and only for products respectively. They did not consider the support for version control and change management sufficiently from method engineering context, i.e. from both side of products and methods.

To solve above issues, this paper discusses a technique to implement a configuration management mechanism in our CAME tool combining with Version Control System for software diagrams, both of which that has been developed before independently [10, 13]. We have two key techniques; the first one is a three-dimensional model to conceptualize the difference between product and method version control [14]. The second is “operation based approach”, where change operations that were performed on an artifact¹ are recorded and are applied in order to recover a current version of the artifact. The rest of the paper is organized as follows. Our CAME tool and Version Control System is introductively summarized in the next section. In addition, we illustrate the details of the issues on version control and change management in method engineering context. In section 3, by using a simple example, we discuss the three-dimensional model for conceptualizing version control and it is very useful to get the solutions to the issues mentioned above. Section 4 discusses how to achieve the change management to maintain consistency on artifacts and clarifies how our technique can solve the issues mentioned in section 2.

2 CAME tool and Version Control System

2.1 CAME tool

Our CAME tool is based on reuse technique similar to the other existing CAME tools such as Decamerone [7], Mentor [15] and MetaEdit+ [9]. Reuse technique is characterized by using reusable method portions, called method fragments or method chunks, which can be extracted from several existing methods. Method fragments are stored in a specific database called method base, and a special engineer, called method engineer obtains suitable fragments from the method base and assemble them into a new and project-specific method. The method engineer, for building a project-specific method, uses a method editor to manipulate method fragments and assemble them into a new method. The method editor is a kind of diagram editor and allows the method engineer to easily edit method

¹ We use the term “artifact” for products and methods.

fragments. The method description is called meta model, and we use Class Diagram to describe a meta model. Our CAME tool generates from a meta model, 1) a diagram editor for supporting inputting and editing products, such as the editor of Class Diagram, and 2) the schema of a repository to which the generated editors store the developed products. Software engineers may develop a model of an information system following the project-specific method, by using the generated editors. The example of the meta model of simplified version of Class Diagram is shown in a Figure 1. As shown in the figure, the method fragment “ClassDiagram” has the concepts “Class”, “Operation” and “Attribute” and all of them are defined as classes on a meta model. These concepts (called method concepts) have associations (called method associations) representing logical relationships among them. For instance, the concept “Class” has “Feature” (a super class of Attribute and Operation), so the association between “Class” and “Attribute” denotes *has* relationship. We simply call both method concepts and method associations method elements.

In addition, we should consider constraints on the products. Suppose that we define the method “ClassDiagram” as shown in Figure 1. In any class diagram (any instance of “ClassDiagram”), we cannot have different classes having the same name. In order to keep consistency of products, we specify this constraint on the meta model, by using OCL (Object Constraint Language). The OCL expression in the right bottom window “CAMEPackage” of Figure 1 represents the constraint of “different names must be attached to different classes”.

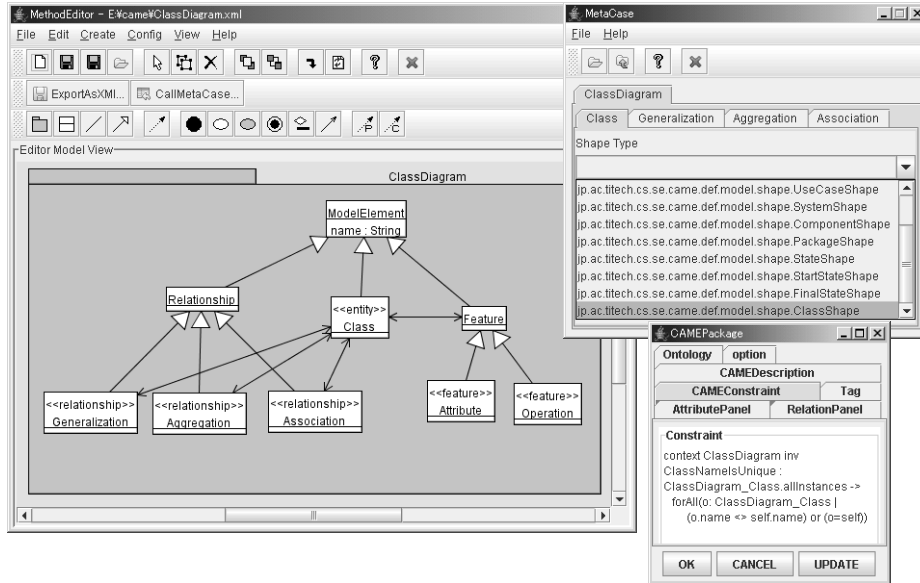


Fig. 1. An Example of Method Fragments

A generated diagram editors deals 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 this information; one is the correspondence of method concepts to the elements of graphs, i.e. nodes, edges and texts within the nodes or on the edges, and another is notational information of the nodes and edges. Suppose that she or he tries to generate a class diagram editor from “ClassDiagram”. The concept Class in the “ClassDiagram” conceptually corresponds to nodes in a graph, while Generalization, Aggregation and Association correspond to edges. She or he provides this information as the stereotypes attached to the method concepts in our CAME tool. The right top window “MetaCase” in Figure 1 includes the information for the generator. The readers can find the stereotypes “<<entity>>” and “<<relationship>>” attached to the classes in the meta model of Figure reffig:sample data. For example, the classes Generalization, Aggregation and Association in the figure have the stereotypes <<relationship>>. The stereo type <<entity>> stands for the correspondence to a node and the <<relationship>> corresponds to an edge. In the example of the figure, an occurrence of Class in a class diagram corresponds to a node from the viewpoint of the graph, while an occurrence of Generalization, Aggregation or Association between Classes corresponds 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, the 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. Basic figures such as figures used in UML diagrams are built-in and their drawing programs are embedded as Java classes into the generator. In the example of Figure 1, the method engineer tries to use a rectangle (ClassShape) for the figure of Class. Our generator produces a diagram editor by embedding the above information and Java classes into a diagram editor framework.

2.2 Scenario Example

In this sub section, we have a following simple scenario of a development as an example, which will be used throughout this paper. It is very useful to clarify the issues of version control and change management in a method engineering context.

A method engineer constructed a new method by assembling Class Diagram (CI#1) and Sequence Diagram (CI#2) of UML² by adding a method association “*instance_of*” as shown in Figure 2. Each meta model can be considered as a unit of configuration management, i.e. configuration item of method level. Following this new method, a software engineer constructs a class diagram of the system to be developed, and then develops the sequence diagrams, each of which defines an scenario of the interactions among objects belonging to the classes appearing in the class diagram. Figure 3 illustrates a part of Lift Control

² Both of the diagrams are more simplified versions for comprehensiveness.

System developed following this method. Each diagram is a configuration item of product level.

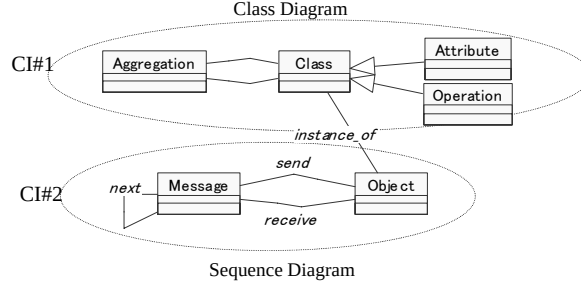


Fig. 2. Assembling Method Fragments

The engineer completes the diagram shown in the left part of Figure 3, and commits it to a repository as the version 0. After that, he adds the object “Door” to the sequence diagram as shown in the right part of Figure 3, and commits it as the version 1. When the engineer adds an object to a sequence diagram, its class should be included in the class diagram in this method M0. If not included, she or he has to add its class to the class diagram to keep consistency between the class diagram and the sequence one. In this example, since “Door” class does not appear in the version 0 of the class diagram, the engineer adds it for the version 1, as shown in Figure 3. The supporting tool hopefully guides the engineer for this kind of change propagation, and change propagations depend on methods and method assembly. The engineer can check-in her or his product after adding “Door” to both of the sequence diagram and the class diagram.

We continue the example. See Figure 4. The engineer finds that Lift Control System has real-time property, and extends the current method so that the engineer can model timing constraints in sequence diagrams. The engineer modifies the meta model of Sequence Diagram (M0 : version 0 of the method) by adding the method concept “Timing Constraint”, and gets a new version 1 (M1). Although we need the version control of meta models, it is the same as the version control of products, because our meta model is represented with Class Diagram as mentioned in section 2.1. The version control of meta models is called “method version control” to distinguish it from usual version control of products (called “product version control”). Now, the engineer continues her or his activities following the new method M1. Since this change to M1 was adding a new method concept only, it has not impacted the current version of the product, version 1. We continue our example further. As shown in the upper part of Figure 4, the engineer adds a timing constraint “b-a < 2 min.” (a lift should be arrived within 2 minutes after pushing a request button to a lift call). Suppose that the engineer return back the method to the older one M0 after that.

Since M0 does not include “Timing Constraint”, the existence of “ $b-a < 2\text{min.}$ ” in the current product causes inconsistency. Thus whenever a current method is changed, we need to check if the new version of the method is consistent to the current version of the product that was made following the older method.

Suppose another change on the method M0 in Figure 2. In the case that the engineer deletes the method association “*instance_of*” and tries to commit it as a new version of the method, what’s happen? As a result, the engineer will get the isolated two methods each of which is the same as the already existing method, i.e. Class Diagram and Sequence Diagram, and this result is not meaningful. Thus we should avoid getting such meaningless versions of the method when we change the used method fragments, and method assembly rules are applicable to prevent from generating meaningless methods [6].

To summarize the above discussions, we can categorize our issues on change management into three; 1) for products, 2) for method fragments and 3) for both. How to solve these three issues will be discussed in section 4.

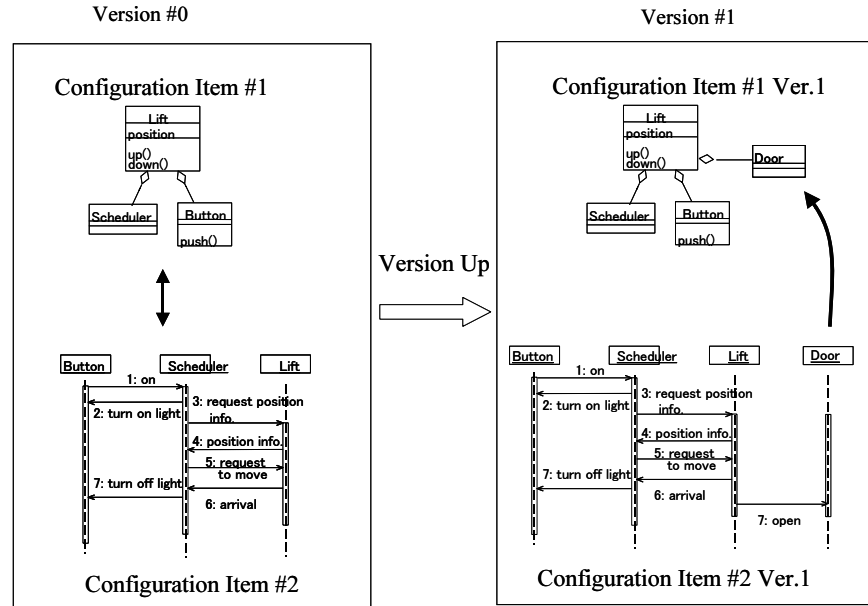


Fig. 3. Lift Control System

2.3 Version Control System

In our version control system, we adopt a technique to store differences between two versions in a repository like CVS [1] and Subversion [2], etc. so that we can

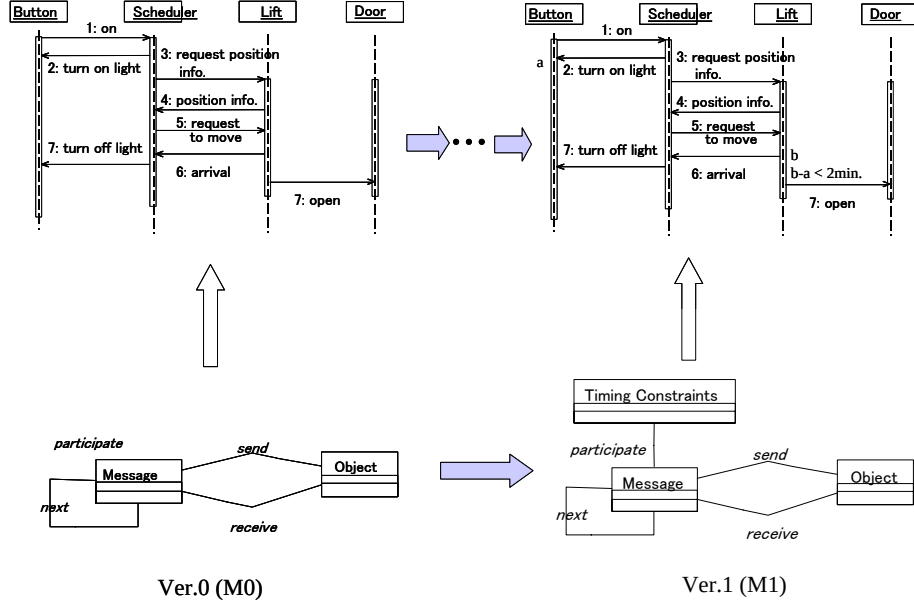


Fig. 4. Version Up of A Method

recover the older versions that were previously produced. The state of the artifact at a certain time is considered as a baseline, and the version control system stores to the repository the difference between this baseline and each version. To extract a difference between two adjacent versions efficiently, we focus on the developer's activities of editing a product by using an editor. In other words, we generate an element of the difference from an execution of an editor operation such as "create" and "delete" a component. The sequence of the editing operations that a developer is performing is captured in real-time during her or 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 CAME tool, which automatically 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. The details of this mechanism was discussed in [10].

Our CAME tool can export the XML document that represent logical information of a diagram in XMI-compliant format [4]³. For simplicity, the representation of differences is based on XMI, and we use XMI.update operations. They are used for informing the differences of XMI-compliant documents when

³ For comprehensiveness, the XML documents in this paper are made simpler rather than real XMI-compliant format.

the documents are exchanged. We have three operations; XMI.add for adding a component to the older document, XMI.delete for deleting an existing component, and XMI.replace for replacing an existing component element with a new one. Figure 5 illustrates how to represent differences with XMI. A software engineer adds “Door” class and an aggregation from “Lift” to it. These change operations performed on the editor are transformed into two XMI.add occurrences and the occurrences are stored as a difference from Version 0 to Version 1. To check-out Version 1 from Version 0, our version control system apply the XMI.add occurrences successively to the XMI document of Version 0, which is a baseline in the sense that the complete document is stored.

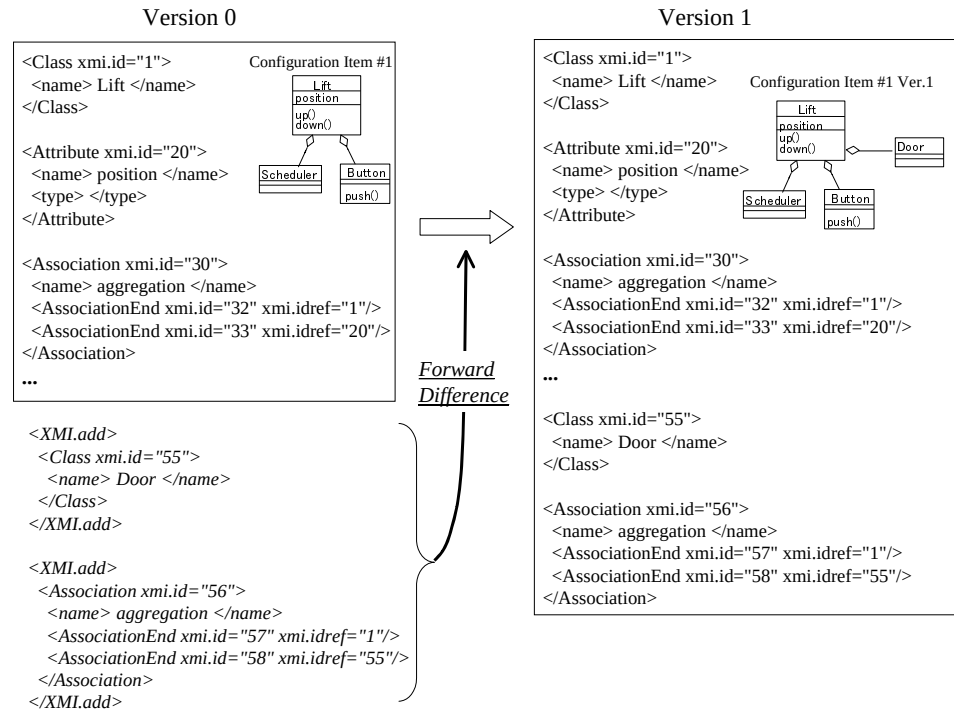


Fig. 5. Representing with XML

Our version control system supports version branching and merging branched versions. Suppose that our software engineer produces a new version Ver. 2 by adding a subclass of “Door” to Ver.1 in Figure 3, while the engineer also creates a branched version Ver.1.1 by deleting the class “Door” from Ver.1. When he tries to merge the Ver. 2 to Ver. 1.1, a conflict occurs. Since Ver. 1.1 does not have “Door” class any longer, adding automatically the subclass of “Door” by

applying the difference from Ver.1 to Ver. 2, is impossible. In this case, our system asks the engineer to take the alternative of adding “Door” by hand to continue this merge operation, or canceling this merge operation. To detect this conflict, each recorded change operation has the pre condition that should be checked before applying it. In XMI.delete, its pre condition is that the object to be deleted should exist in the product. In the above example, the operation “<XMI.add> <Association xmid.id =...> <name> Generalization </name>...</XMI.add>” (adding an association Generalization from Door class to a subclass) requires a source object and a destination of the association as a pre condition, i.e. “Door” is required to execute this operation. Pre condition are automatically generated and attached to change operations to be stored as a difference. Pre conditions maintain consistency not only for merging branched version but also for change propagation, as mentioned later in sections 4.2 and 4.3.

3 Conceptual Model for Version Control

In this section, we show a three-dimensional model to conceptualize our version control technique [14] and how to use it. We have “products” and “method fragments” as targets of version control, and each target consists of configuration items. Thus, we can consider in a version space three axes; product, method (fragment) and configuration item as shown in Figure 6. Each lattice point in the figure represents a version of a product to be managed.

In our version control system, an engineer has working spaces at his local site, and performs check-out and check-in operations between her or his working space and a repository. When the engineer checks out from the repository the version n of a product which has been developed by method M , a working space for version $n+1$ is allocated at her or his local site and an editor for M is invoked. The version n of a product is loaded into the working space. The engineer uses the editor to modify the version n , and after completing the modification, he or she stores it as version $n+1$ into the repository (check-in). A working space is generated allocated method by method. In the case that the engineer uses the methods $M0$ and $M1$, both a working space on $M0$ and a working space on $M1$ are generated. Note that our repository has two levels: one is for storing products and the other is for meta models.

Following the scenario of Figures 3 and 4, consider what operations our engineer performs on our version control system. The engineer’s activities are illustrated in Figure 7. The engineer selects the method $M0$ and generates an empty working space by using “new” command at first. As shown in Figure3, the method M is the result of assembling Class Diagram and Sequence Diagram, and the engineer develops two types of diagram by using each graphic editor (2:input & edit). Let the two diagrams be $C0$ and $S0$ respectively. The engineer checks in them to the repository (3:check-in) and they are stored as version 0 ($P0$). Continuously, the engineers adds the “Door” object to the sequence diagram $S0$ (4: edit) and gets the version 1 ($S1$). If the engineer tries to check in it to the repository, she or he fails because the current $P0$ is not satisfied with the constraint “for each object in a sequence diagram, its class is included in a

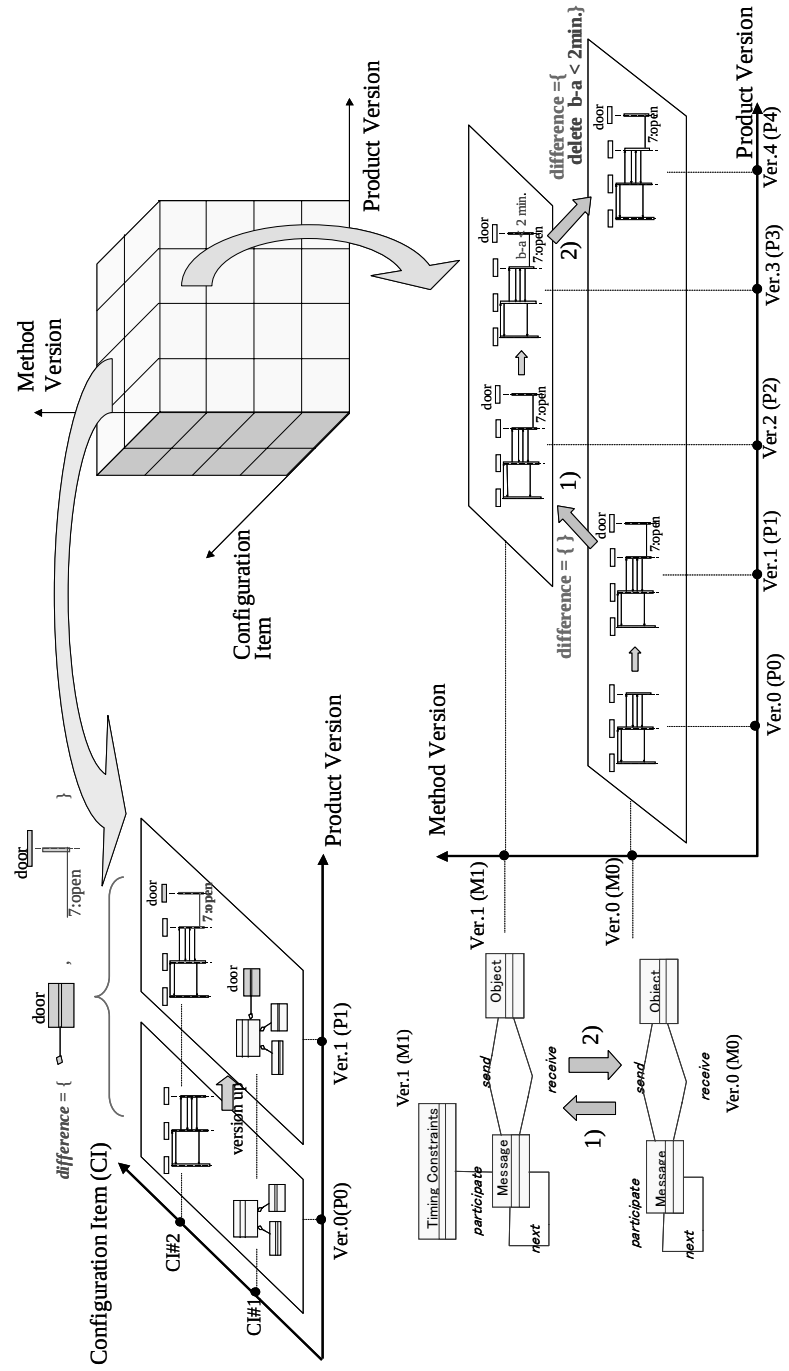


Fig. 6. Three Dimensional Model

class diagram”. To get consistency, the engineer adds the Door class to the class diagram C0 and successfully checks in it (5: check-in). The new product comes in the repository as version 1 (P1).

Furthermore, the engineer tried to extend the method M0 to M1 as shown in Figure 4, and checks out M0 from the meta-level part of the repository (6: check-out). The engineer can have a working space for constructing M1, and M0 is loaded to the space. By using a method editor, as shown in Figure 4, the engineer adds the method concept “Timing Constraint” to M0 (7:edit) and then checks in it as version 1 (M1) to the repository (8:check in). To continue his task by using the new version M1, he creates an empty working space for P2 on M1 (9: new), and checks out P1 to this space (10: check-out). After that, the engineer adds timing constraint “b-a < 2 min.” (11: edit) and checks in the resulting product (12: check-in). This product is registered into the repository as version 2 (P2).

Next, suppose that for a reason, the engineer comes to want to return the used method back to the older version M0. The engineer tries to import M0 at the current working space (13: import). When importing M0, the system checks consistency of the current product to M0 and the import operation is succeeded if the consistency check is passed. In our example, since the difference between M0 and M1 includes `<XMI.delete> ... “Timing Constraint” ... </XMI.delete>` and the current product has its instance “b-a < 2 min.”, the engineer is notified of the inconsistency. The engineer deletes “b-a < 2 min.” according to the notification and then imports M0 again. Now, the engineer succeeds in importing M0 and checks in the current product as version 3 (P3) to the repository (14: check-in). Figure 6 includes projections of this simple scenario in the 3 dimensional cube, and the readers can trace a trajectory of the engineer’s activities in the cube.

4 Solving Issues on Change Management

4.1 Change Propagation on Products

Consider again the example scenario in section 2.2. Our software engineer added the “Door” object to the sequence diagram and checked in it to the repository, as shown in Figure 3. However, this adopted method consisting of Class Diagram and Sequence Diagram requires the addition of “Door” class to the class diagram in order to maintain consistency on the product. This is typical change propagation on configuration items in product level. The supporting tool hopefully guides the engineer for this kind of change propagation, and it depends on methods and method assembly. In our CAME tool, we can specify the constraints with OCL as shown in the right bottom window of Figure 1. In fact, we put the constraint “for each object in a sequence diagram, its class must be included in a class diagram” with OCL when assembling Class Diagram and Sequence Diagram into the example method. We can realize this type of change management on configuration items by means of consistency checking using OCL evaluator.

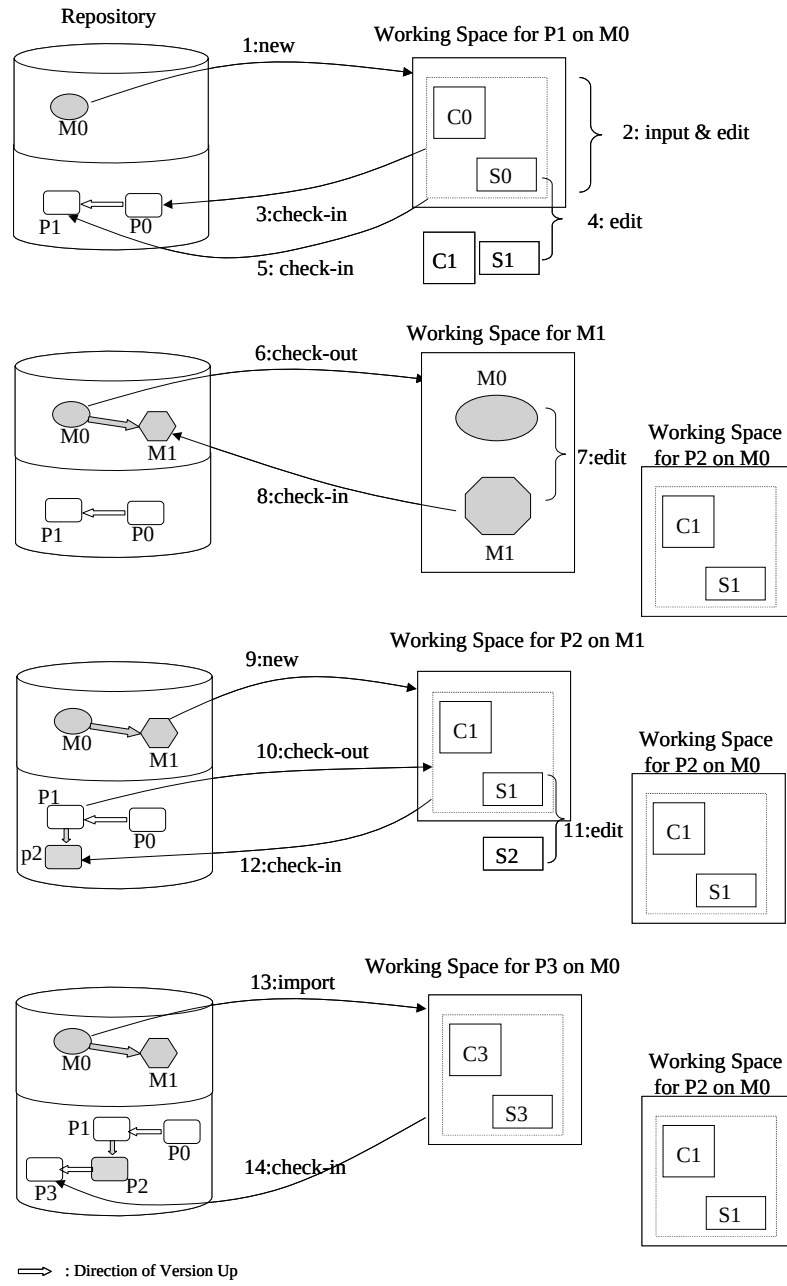


Fig. 7. Version Control System

4.2 Change Propagation on Methods

As for change management on method fragments, we can consider two categories. The first one is quite similar to the consistency checking on configuration items of product level, which was mentioned in the section 4.1. Since our method fragments are defined as class diagrams and activity diagrams, consistency checking on them is possible by using constraints written with OCL in the same way as consistency checking on products. The constraints are not defined by method engineers, unlike product level, but as method assembly rules in advance. For example, we have a method assembly rule “at least one method concept and/or method association that connects the method fragments to be assembled should be newly added”, which says that when we assemble method fragments, we should connect logically them by using newly added method elements [6]. Suppose that our method engineer deletes a method association “*instance_of*” between “Class” of method fragment “Class Diagram” and “Object” of “Sequence Diagram” in Figure 2, as illustrated in section 2.2. This deletion operation violates the above method assembly rule and causes logically isolation of these two method fragments in the resulting method. Checking consistency is performed by using the method assembly rules represented with OCL, and it is the same technique in the section 4.1.

The second one is the propagation to the other methods that have used the changed method fragments. See Figure 8 and suppose that we have two methods M#1 and M#2; M#1 are composed from Class Diagram and Sequence Diagram, and M#2 are from State Diagram and Sequence Diagram. The method engineer updates the fragment MF#3 (Sequence Diagram) by adding “Timing Constraint” concept as shown in Figure 4. After this version-up, what happens to the existing methods M#1 and M#2 of version 1? It is desirable that M#1 and M#2 are automatically updated to their newer versions having the new Sequence Diagram fragment Ver.1. The difference from Ver.0 to Ver.1 of Sequence Diagram is automatically applied to Ver.0 of M#1 and M#2 so as to get their newer versions Ver.1. As a result, the method engineer can get the newer versions that have “Timing Constraints” concept in Sequence Diagram parts in M#1 and M#2. During the application, the pre conditions of the change operations included in the difference are verified so as to avoid inconsistency. It is the same way as merging branched versions mentioned in section 2.3. After finishing the application, the generated newer versions, i.e. Ver.1 of M#1 and M#2, are verified whether method assembly rules are satisfied or not.

4.3 Change Propagation between Products and Methods

Consider again the example scenario of sections 2.3 and what we should do to maintain consistency, when the method is changed back from M1 to M0, i.e. deleting “Timing Constraints” concept, as shown in Figures 6 and 7. By using the forward difference for the version-up from M0 to M1, we can get the backward difference from M1 to M0 as follows.

```
<XMI.delete>
  <Class xmi.id="102">
```

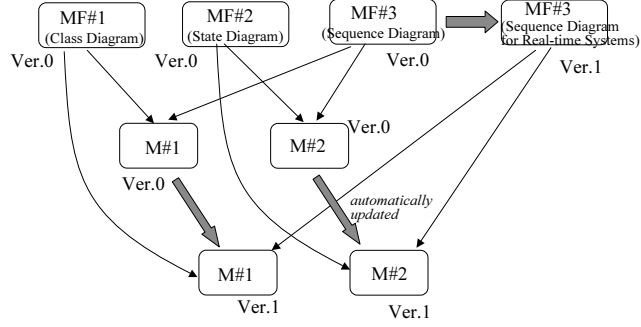


Fig. 8. Change Propagation on Methods

```

    <name> Timing_Constraints </name>
  </Class>
</XMI.delete>
<XMI.delete>
  <Association xmi.id="103">
    <name> aggregation </name>
    ...
  </XMI.delete>

```

It is easy to automatically get the above difference, by replacing the occurrences of “add” with “delete” and vice versa in the recorded difference from M0 to M1. In the case that the method engineer deletes a method concept or association from a method fragment and commits it as a new version, we can get a difference including “XMI.delete”. All that we should do for consistency checking to the current method are to look for “XMI.delete” in the difference from the last version to the current one, and to extract the method elements included in XMI.delete. And then, we detect the components whose types are the extracted method elements. In this case, we can extract the method element “Timing Constraints” appearing in the above XMI.delete fragment, and then can detect the components whose type is “Timing Constraints” in the sequence diagram. If the components are detected in the current version of a product, our tool informs the engineer that inconsistency occurs on account of changing the method. The technique of detecting this kind of inconsistency is to focusing on the occurrences of XMI.delete in the difference of a method change.

5 Conclusion and Future Work

This paper discusses the problems of configuration management, especially version control and change management in method engineering environments, and proposed an integrated technique to solve them. In particular, in section 4, we clarified various types of change propagations in a method engineering context,

and showed that we could solve their issues by our proposed technique. And we discussed the CAME tool where this technique is embedded.

Although we have implemented basic commands mentioned in section 4 and embedded our CAME, we need more functions, in particular browsing in a repository, displaying the status of products (consistent or not, the newest version or not, etc.), and retrieving a specific version not only by using version number, to make it more practical. And more case studies are necessary to assess our technique and the CAME tool together with version control functions. The support for cooperative tasks by a team is also one of the future works.

References

1. Concurrent Versions System. <http://www.cvshome.org/>.
2. Subversion. <http://subversion.tigris.org/>.
3. The Coral Metamodeling Toolkit. <http://mde.abo.fi/tools/Coral/>.
4. XML Metadata Interchange. <http://www.omg.org/>.
5. C. Bock. UML without Pictures. *IEEE Software*, 20(5):33–35, 2003.
6. S. Brinkkemper, M. Saeki, and F. Harmsen. Meta-Modelling Based Assembly Techniques for Situational Method Engineering. *Information Systems*, 24(3):209–228, 1999.
7. F. Harmsen. *Situational Method Engineering*. Moret Ernst & Young Management Consultants, 1997.
8. R. Keller, J.-F. Bedard, and G Saint-Denis. Design and Implementation of a UML-Based Design Repository. In *Lecture Notes in Computer Science (CAiSE2001)*, volume 2068, pages 448–464, 2001.
9. S. Kelly, K. Lyytinen, and M. Rossi. MetaEdit+ : A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In *Lecture Notes in Computer Science (CAiSE'96)*, volume 1080, pages 1–21, 1996.
10. T. Oda and M. Saeki. Generative Technique of Version Control Systems for Software Diagrams. In *Proc. of the 21st IEEE Conference on Software Maintenance (ICSM'05)*, pages 515–524, 2005.
11. J. Ralyte, C. Rolland, and R. Deneckere. Towards a Meta-tool for Change-Centric Method Engineering: A Typology of Generic Operators. In *Lecture Notes in Computer Science (Proc. of CAiSE'2004)*, pages 202–218, 2004.
12. N. Ritter and H.-P. Steiert. Enforcing Modeling Guidelines in an ORDBMS-based UML-Repository. In *Proc. of International Resource Management Association Conference (IRMA2000)*, pages 269–273, 2000.
13. M. Saeki. Toward Automated Method Engineering: Supporting Method Assembly in CAME. In *Engineering Methods to Support Information Systems Evolution (EMSISE'03 in OOIS'03)*. <http://cui.unige.ch/db-research/EMSISE03/>, 2003.
14. M. Saeki and T. Oda. A Conceptual Model of Version Control in Method Engineering Environment. In *Proc. of CAiSE Short Paper 2005*, pages 89–94, 2005.
15. S. Si-Said, Rolland C., and G. Grosz. MENTOR : A Computer Aided Requirements Engineering Environment. In *Lecture Notes in Computer Science (CAiSE'96)*, volume 1080, pages 22–43, 1996.