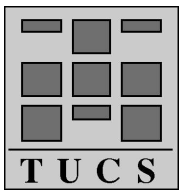
Issues on the Design of an XML-Based Configuration Management System for Model Driven Engineering

Marcus Alanen and Ivan Porres

TUCS Turku Center for Computer Science
Åbo Akademi University
Lemminkäinsenkatu 14A, FIN-20520 Turku, Finland
e-mail:name.surname@abo.fi



Turku Centre for Computer Science
TUCS Technical Report No 567
November 2003

ISBN 952-12-1258-6
ISSN 1239-1891

Abstract

We review the central concepts required from a project and configuration management system for MOF-based models. The necessary features for a model repository and MOF framework are given, as well as suggestions of several algorithms and components.

Keywords: Model management, Model versioning, Model repository, MOF

TUCS Laboratory
Software Construction Laboratory

1 Introduction and Motivation

In this article we study how to store, manage and organize large models during the lifetime of a software project. The first generation of UML editors used to store a whole model as a single file. This approach is good enough in a waterfall-like development process where one single software designer is the only person in charge of model development. This approach assumes that once a model is ready, it can be printed and distributed to the programmers as documentation since there will not be major changes. Programmers use the model as a reference design or blueprint for the code to be developed, but the model is not updated any longer. In this scenario, software evolution and maintenance reverts over to the program source code, not to the UML model.

However, this approach is not satisfactory if we plan to use UML models instead of source code as the main and most important description of our software. This requires that any model should be always up to date. In this context, there will be different developers working simultaneously on the same UML models. There will be different versions of the same model, targeted to different platforms or customer requirements, and evolution and maintenance will be carried out over the UML models. This implies that we need to use a proper configuration management system to keep track of our software models.

Configuration management is a well-studied topic in the literature and there are many tools available on the market. It involves several different subtopics such as version control as well as change, build and release management. At the same time, configuration management is a key element in the management of any software development project. It is possible to construct a self-made system using a combination of open-source tools such as CVS, autoconf, make and Tinderbox, or to use complete commercial solutions such as IBM Rational ClearCase.

However, most of the existing tools are designed to manage either program code or informal documents in natural language. The question now is if we can use existing configuration management systems to keep track of evolving UML models or if we need new tools and methods customized to the idiosyncrasies of the Object Management Group (OMG) standards. The objective of this article is to raise different issues that appear when we try to use inappropriate methods and tools to manage UML models while discussing possible alternatives that comply with the existing standards.

1.1 Modeling Languages and Metamodels

According to the OMG standards, the information stored in a UML model is organized internally according to a metamodel. A metamodel describes the abstract syntax of a modeling language. Each class in a metamodel describes a model element, i.e., a concept or abstraction in our modeling language. Each class may have a number of attributes. An association connecting two classes represents a symmetric relation between these elements.

We can illustrate these concepts in a small modeling language of our invention that is much simpler than UML. Our example language is called FSM and is a language for describing finite state machines. A state machine has a finite number of states and transitions. Each transition connects two states and it can be triggered by a token. The set of tokens in a state machine is called the alphabet. One or more states may be marked as accepting states, while one of the states is marked as initial. These concepts are described as a class model in the left side of Figure 1. We call this kind of diagram a metamodel. This diagram is similar to the metamodels shown in the OMG UML standards.

In our example language, the fact that each state machine has an initial state is represented by the association named initial. We use the generalization relationship to define a model element as a specialization of other model elements. In our metamodel, an accepting state is a specialization of state.

Figure 1 shows an example model in the FSM language. The model is represented using two different notations. The diagram at the right uses a syntax that is specific for our language for finite state machines. Most designers would prefer this notation since it is a fully visual language where each concept is described using a different icon.

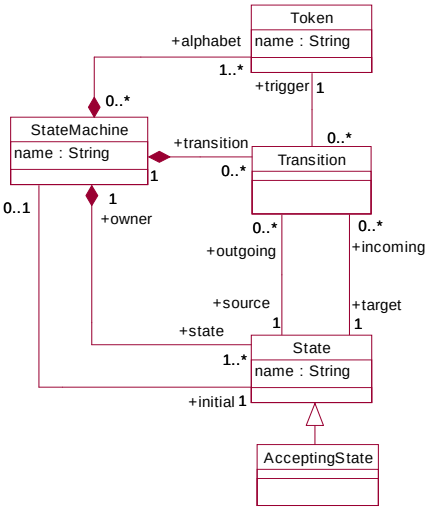
However, we can also represent the same model as an XMI document. XMI is an OMG standard [9] for model interchange. It is based on XML and can be used to represent any modeling language, i.e., it is not limited to UML. XMI is the preferred notation to exchange models between programs since XMI documents are portable and easy to parse.

There is a one-to-one mapping between a model described as a diagram and a model described as an XMI document if and only if both representations conform to the same metamodel. The UML metamodel is defined in a language called the Meta Object Facility (MOF). MOF is also defined as an OMG standard, and it can be use to define many different modeling languages, e.g. there is nothing specific to UML in MOF. In this article, we assume that the models representing our software are described as a MOF metamodel. In this sense, this article is not specific to UML but to MOF. However, UML is the largest, most used and best known MOF application, so we will use the UML to illustrate our findings.

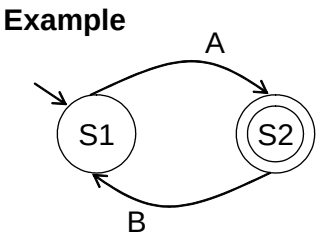
1.2 A Model-Based Configuration Management System

A configuration management (CM) system is based on a central repository that contains all artifacts relevant to a software project. We define a model-based configuration management system as a CM system where the project artifacts are structured logically as defined in a metamodel such as the UML modeling language. Besides the repository, a CM system is composed of at least three other components: a version control system, a change control system and a build system.

A version control system should store and keep track of different versions of each artifact or document created in a project. A version of a model can represent a different design solution or a different implementation of the same design in a



FSM Metamodel



Example Model
in a Concrete Notation

```
<XMI xmi.version='1.1'>
<XMI.header>
  <XMI.metamodel xmi.name='FSM' xmi.version='1'/>
</XMI.header>
<XMI.content>
  <FSM:StateMachine xmi.id='if564' initial='i5044'
                    name='Example'>
    <FSM:StateMachine.alphabet>
      <FSM:Token xmi.id='i8e5d' name='A'></FSM:Token>
      <FSM:Token xmi.id='i606b' name='B'></FSM:Token>
    </FSM:StateMachine.alphabet>
    <FSM:StateMachine.state>
      <FSM:State xmi.id='i5044' name='S1'
                 stateMachine='if564'>
...
```

Example Model as an XMI Document

Figure 1: Example Model in the FSM Language

given platform.

The second component is a change control mechanism that defines who can introduce new artifacts and new versions of an existing artifact in the repository and how these changes are reviewed and approved. In larger projects, it may be of interest to restrict the modification of classes or components that are well designed, implemented and/or tested. In safety-critical system, the models should not be changed without a formal change procedure.

Finally, a build system creates and recreates executable programs based on the artifacts in the repository. A common approach in many projects is to create nightly builds or even continuous builds automatically. Also, a build system can run a set of unit and integration tests automatically and generate status reports that are distributed among the developers.

CM systems are usually distributed systems that allow different developers to work simultaneously on the same project. In this case, the repository resides on (at least) one server and the client computers read and update parts of the repository as needed. We would like that the communication between the repository and the client is based on open standards so we can seamlessly use tools from different vendors. The OMG standards propose XMI [9] as the standard model interchange format and we can expect that the repository server and the client will use XMI as a common format. However, the server may store the models in the repository using a different format or use auxiliary index files to find the information inside a XMI file quickly.

In the rest of the article we will discuss which issues appear in a model-based CM system. In order to implement a repository, version, access and build system we need to be able to perform several basic tasks on the elements of a model. First of all, we must be able to uniquely identify elements in a model, and find elements according to a specific criteria. Then, we must be able to further query about the data of the elements. Finally, we must be able to change that data. Together, these operations can be used to calculate differences between elements, track element evolution, merge element data and resolve merge conflicts, restrict read and write access to specific parts of a model, transform elements and enforce a process upon the development of the models.

2 Basic Model Management

The most fundamental requirement for a CM system is to be able to uniquely identify the elements stored in it. One of the most widely used mechanisms to identify an element in a repository is to use a hierarchical naming schema. The file name C:\My Documents\UML\evolution.tex or the Java class name java.util.Iterator are examples of hierarchical names. In UML we can create similar names using two colons as a separator. A class named Person inside the package Sales can be referred to as Sales::Person. Hierarchical names are intuitive and easy to use. However, there are two problems with using this mechanism to identify elements in a

model.

First, if we rename an element in the model, we loose the linking between the current and the previous version. As an example, if we rename the class Person to Customer, there will be nothing in our repository that would tell us that Sales::Customer is actually derived from Sales::Person. The solution would be to somehow add this missing information to the repository. However, there is a another problem: Not all the UML model elements have proper names. For example, generalization relationships are never named. The same applies to transitions in a statechart, links in a sequence diagram and many other minor but equally important elements.

The solution is provided by the XMI standard itself. The standard states that each element in a model may have a Universally Unique Identifier (UUID) (pp. 1-3 of [9]). An example UUID is the string:

DCE:2fac1234-31f8-11b4-a222-08002b34c003

UUIDs are assigned the first time that an element is exported to an XMI document. Later, any standard-compliant open tool that imports the XMI document should not change or remove the assigned UUIDs.

UUID strings [4] are assumed to be globally unique. They are based on a 128-bit pseudorandom number generated from the physical address of the network interface in the host running the tool and the tenths of microseconds elapsed since the Gregorian reform (15 Oct. 1582). The UUID strings are long and too complex to be generated by hand but this is not an issue for the user since the UML tools should take care of this aspect. Unfortunately, many of the existing UML tools do not generate or even preserve UUID strings. To check this we can perform a simple test. Use your favorite CASE tool to generate a small model (it can be empty) and save the model in an XMI file. Edit the XMI file with a text editor and add a UUID identifier to the Model element. For example:

```
<UML:Model xmi.id = '122'
           xmi.uuid = 'DCE:2fac1234-31f8-11b4-a222-08002b34c003'
           name = 'Example Model' isSpecification = 'false'
           isRoot = 'false' isLeaf = 'false'
           isAbstract = 'false'>
```

Then save and close the file in your text editor and load it in the CASE tool. The modified XMI file should be imported without problems. Export the model again to the XMI format and open the XMI file with your text editor. Examine the line defining the Model element. The UUID should be there, unaltered. If the CASE tool has modified the UUID string or has removed it completely then it does not comply with the XMI standard for open tools.

UUID strings allow us to differentiate between two instances of the same element and two elements that are similar. We consider that two model elements of

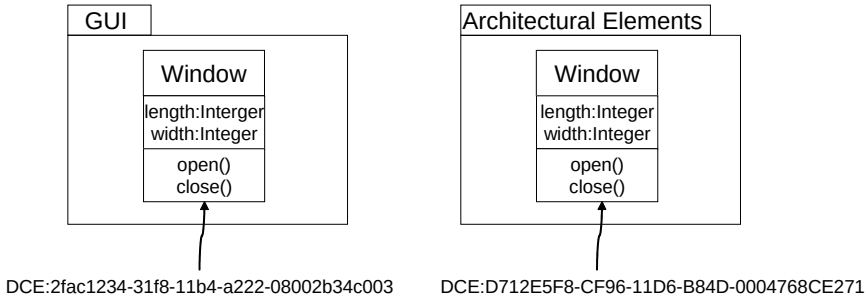


Figure 2: Unique Identification of Elements using UUID

the same type and with similar properties are still two different model elements if their UUID strings are different. An example of this is illustrated in Figure 2. This figure contains two classes that have the same name and properties but that represent two different abstractions from two different problem domains.

This distinction is fundamental to implement model management operations like comparing two models, merging two models into one or duplicating (parts of) a model. If we merge the two models represented in Fig. 2 into one, we want to keep two different classes named Window, since they represent two different things.

In other cases, model elements with the same name do actually represent the same abstraction. One typical example are the standard classes predefined in a programming language such as Integer or java.util.Iterator. The two models from Figure 2 implicitly contain two classes named Integer, that probably are the same concept. We can solve this problem by assigning a predefined identifier to standard elements such as the libraries of programming languages. In fact, the Java Object Serialization Specification states that each (serializable) Java class has a unique 64-bit integer used to uniquely identify the class in a stream. We could derive the 128-bit UUID from the 64-bit Java identifier. However, the XMI standard does not describe how to do this. Also, other programming languages like C++ do not have assigned identifiers for their library elements. A possibility would be to identify such elements by name, for example c++.std.iostream.cout, but this is exactly what we are trying to avoid by using UUID strings! The solution may be to standardize a method to create UUID strings based on the name and signature of these classes. This way it could be possible to generate automatically UUID strings for the standard library of languages such as C++.

The next question is what happens when we have two instances of a model element, with the same UUID but different properties. Since we assume that UUIDs are unique, we have two versions of the same element. In this case, we want to be able to detect that the element has been changed and to calculate what actually has been changed.

```
file1.xml:
 <UML:Model xmi.id = '122' name = 'Example Model'/>

file2.xml:
 <UML:Model name = 'Example Model' xmi.id = '122'/>
```

Figure 3: The same model in XMI as two different ASCII files

2.1 Difference Between Models

Computing the differences between two different models or two versions of the same model is a fundamental operation in a CM system. This basic operation allow us to define the evolution of a model as the sequence of differences between two consecutive versions of the model.

The implementation of this operation may seem trivial. Since XMI files are basically text files, we could try to use a tool designed to compute differences for line-based text files such as source code to analyze our models. The UNIX programs diff and patch are two fine examples of these tools. The problem is that line-based tools will detect false changes in a model. For example, file1.xml and file2.xml as shown in Figure 3 represent the same UML model, probably generated by two different UML editors. However, a line-based tool such as diff considers these files different.

The next possibility is to use an XML-based tool. Such a tool should be aware that the previous example represents the same document. Still, an XML-based tool is not aware of special features of a metamodel such as whether some elements in a model are ordered or not. For example, the order in which the classes in a package are defined is not relevant in a UML model. Considering this, file3.xml and file4.xml as shown in Figure 4 represent the same UML model, although they are two different documents at the XML level. In other cases, such as the definition of the parameters in a method of a class, the actual order of the parameters is relevant. So, using an XML-based tool that simply ignores the order in which elements are defined is not a solution to this problem either. The ordering information of metamodel associations is only available in the metamodel and will be ignored by a generic XML tool.

As a consequence of the previous discussion, we can only compute differences between two models by using a tool specifically designed to handle XMI models and only when the tool has access to the metamodel used in the model. We have presented the basic algorithms for such a tool in [2].

```
file3.xml:
 <UML:Model xmi.id = '122' name = 'Example Model'>
    <UML:Namespace.ownedElement>
      <UML:Class xmi.id = '123' name = 'Customer'>
      <UML:Class xmi.id = '124' name = 'Product'>
      ...
file4.xml:
 <UML:Model xmi.id = '122' name = 'Example Model'>
    <UML:Namespace.ownedElement>
      <UML:Class xmi.id = '124' name = 'Product'>
      <UML:Class xmi.id = '123' name = 'Customer'>
      ...
```

Figure 4: The same model in XMI as two different XML documents

3 A Model Repository

The task of a model repository is to store successive versions of a model and retain old versions. A simple model repository can store each version of a model as a different file containing the model as an XMI document. In such a system, the file name can be used to identify each version of a model in the CM system. Access control to the repository is managed by the access control mechanism of the filesystem.

This simple repository is too coarse-grained for most practical uses. We may want to use the CM system to keep a history of the evolution of a model through the whole development cycle. In this case, it is important that we are able to identify, version and retrieve each individual element such as a package or a class in a model. While a filesystem still could be used for all this, it is not efficient and we can quickly run into problems with respect to e.g. atomicity and concurrency. Although it can be noted that filesystem design has recently begun to evolve in a direction involving frameworks to solve these problems, a more flexible and robust method is desired.

Storing models into a database provides a solution. A database can set arbitrary rules for access, modification and retrieval, also accomplishing it in a safe manner due to the ACID properties: atomicity, consistency, isolation and durability. Additionally, it is very easy to support even arbitrary metadata of the models. There are, naturally, different ways to accomplish this, even though most designs revolve around a structure similar to the one shown in Figure 5 [1].

As an example, a relational database could separately store each individual version of every element. Then, a version of a model is a collection of versioned elements. How well-suited a relational database is to store hierarchically structured information, which models are, remains to be seen. The upside is that relational

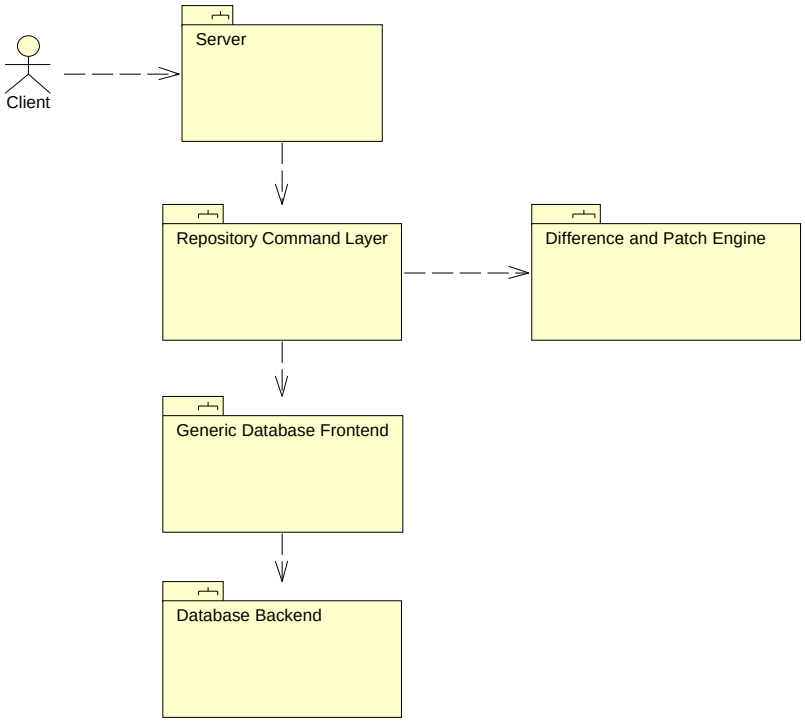


Figure 5: Basic Model Repository Design

databases have been researched very thoroughly and industry has greatly invested in creating highly scalable and efficient products. The downside might be that model information is inherently object-oriented and perhaps does not map naturally into a relational domain.

The advent of XML has started research in databases particularly suited for storing XML documents. XML repositories are very similar to object-oriented databases (OODBs), and share their benefits and ills. Among the benefits are much more flexible arrangements of data, ways to manipulate that data and more complicated queries. However, current technology does not scale as well as relational databases; especially query optimization is not as well-known as in the relational database field. Using an XML database itself could be a great advantage, but until technology catches up, it cannot be deployed for large-scale projects.

In any storage mechanism, the quality of implementation of the repository dictates further characteristics. Where the repository fails in its goals, supporting an extensible architecture using client-side or server-side scripting comes into mind.

3.1 Finding Elements in a Repository

Most of the times a client will not be interested in all the elements in a model but only in a subset of them. The problem is that a client might not know the name or

the UUID of a certain model element in which it is interested. There are two main solutions to this problem: one is to let a client seek elements in the model and the other is to implement a query language.

In the first solution the server should provide two simple interfaces: one service, named getRoot, returns the root element in a model, while the second service, seek, accepts a UUID as a parameter and returns the model element associated to it. The former is for cases where the client wants to traverse the model as it wishes, and the latter is for cases where the client knows which UUID it wants. In this solution, deciding which elements are required is a responsibility of the client.

Another solution is to use a query language, something akin to the SQL in the world of relational databases. In this case, the client will send a query as a text string to the repository that will evaluate the query against all elements in the model and return those who satisfy it.

We can use different alternatives as a query language. OCL [14] is used in the UML metamodel to define additional constraints over valid UML model elements, but it can also be used as a query language. As an example, the following query will return all the subclasses of a class named Customer:

```
self.oclIsKindOf(Class) and
    self.generalization.exists(g: g.parent.name="Customer")
```

Unfortunately, most UML practitioners are not familiar with OCL. Also, the current OCL parsers are not as optimized as the existing database engines. This is due to the fact that we still do not know which are the most common queries that should be optimized. Finally, we would need to extend the current OCL standard with queries to retrieve version information so we can perform queries against the version history of the repository such as

```
self.name="Customer" and self.lastEdited<"1 Oct".
```

An alternative to OCL is to use a query language based on XML, such as XQuery [13]. However, the syntax of XQuery and other XML-based languages is too cumbersome. Currently, parsing and compiling technology is so advanced and desktop computers so powerful that there is no reason to obfuscate the syntax of a computer language to make it easy to parse by a computer. Also, this approach does not solve the need to know how the model information is arranged in the UML metamodel in order to create a complex query.

3.2 Communication Protocol

XMI as such does not define a protocol for transferring models over a network, only the encoding of a model. In the interest of software compatibility, common standards ought to be defined. Special interest groups, separate from OMG, are advancing the state of the art of distributed authoring, and are creating official Internet standards to fill this void. Good examples are the IETF WebDAV

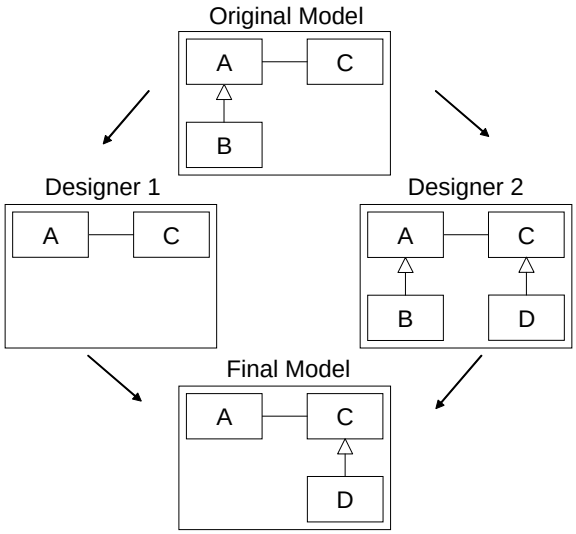


Figure 6: Union of Multiple Versions of a Model

and Delta-V working groups, which have defined e.g. "HTTP Extensions for Distributed Authoring – WEBDAV" (RFC 2518) and "Versioning Extensions to WebDAV" (RFC 3253) to ease communication in a distributed development environment [6, 5]. These or similar standards can be used to define standard protocols between a model repository and the client tools, such as a UML editor.

4 Version Control

A version control systemkeeps track of what has changed in different versions of a configuration item. It also can combine different changes into a new item. In the context of a model-based CM system, version control is provided by the possibility to calculate the difference between several versions of a model and to combine a difference between models into another model. As an example, lets assume that the original model shown at the top of Figure 6 is edited simultaneously by two developers. One developer has focused his work on the classes A and B and decided that the subclass B is no longer necessary in the model. Simultaneously, the other developer has decided that class C should have a subclass D. The problem is to combine the work of both developers into a single model. This is the model shown at the bottom of Fig. 6.

During the rest of this article, we will denote Δ as describing a difference between two models, and $\Delta(M)$ as applying a difference to a model M, returning a new model. The example described earlier can be decomposed into three tasks. Calculate the difference Δ_1 between the model from Designer 1 and the original

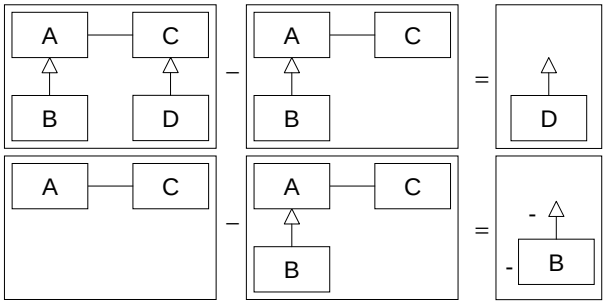


Figure 7: Difference of Models

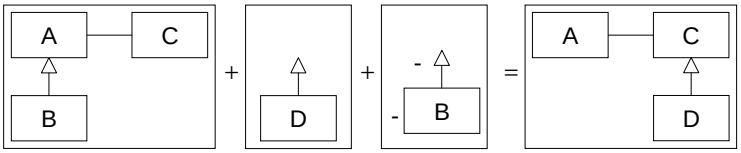


Figure 8: Union Based on Differences

model (Bottom of Fig. 7), calculate the difference Δ_2 between the model from Designer 2 and the original model (Top of Fig. 7) and finally, merging the original model with the two differences (Fig. 8). The result of a difference is not always a model, in a similar way that the difference between two natural numbers is not a natural number but a negative one. An example of this is shown in the bottom part of Fig. 7. In this case, the difference of the models contains *negative* model elements, i.e., elements that should be removed from a model.

The way the difference calculation can be done is shown in Figure 9; either of the Δ_i is modified according to the other, and then applied after the other has been applied. Unfortunately this leads—akin to ordinary line-based repositories—to conflict situations, which will be discussed next.

As a conclusion, the best solution to implement a version control system for models is to be aware of the features of the metamodel in question, and bring interoperability between tools with XMI. It is of essence that the version control system understands its contents, since this allows additional operations to be performed on the stored models, such as searching of model elements, difference and merge calculation of models.

4.1 Conflict Resolution in a Merge

There are several cases where merge conflicts are a fact and manual resolution is required. Modifying the same attribute or the same ordered slot easily creates such situations. For association slots, the opposite slot must also be kept in synchronization. The extreme case of deleting an element even though another difference

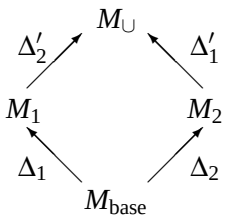


Figure 9: The principle of calculating the union of two models, given their base model. Either difference is modified according to the other one, and then applied.

merely modifies it slightly leads to a complex question; which difference should be prioritized? Further work in this area is clearly required as automatic conflict resolution can be considered important in a modeling framework.

A pure XMI-based approach is not as thorough as one with knowledge of the metamodel. This is due to the fact that XMI considers all properties to be ordered, even though some of them are not. A great number of seemingly conflicting cases can be resolved automatically if the property under modification is actually unordered.

As such, conflict resolution has three distinct steps [2]: 1) A metamodel-independent resolution step, 2) a metamodel-dependent step where the conflict resolution algorithm takes the metamodel of the elements and their well-formed rules into consideration, thus providing automatic resolution where possible, and 3) a step for manual resolution by the developer. Naturally, the work to be done should become smaller for each step for this to be a viable mechanism.

The second step in the conflict resolution mechanism can also include specific heuristics for conflicts depending on the metamodel. A prime example is diagrammatic information, as the diagram elements themselves do not have any semantic meaning, so the features of the diagram elements are not nearly as correctness-critical as the underlying model. For example, conflicting diagram element coordinates on the diagram canvas can more or less be completely ignored by modifying Δ_2 suitably. Clearly, there is a strong need for metamodel-specific resolvers.

The schema in Figure 10 summarizes a merge system for models. The difference under modification, Δ_2, passes through several filters which modify it to better fit $\Delta_1(M_{\text{base}})$. Obviously, all possible mechanic resolution mechanisms should be tried before manual resolution is used.

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

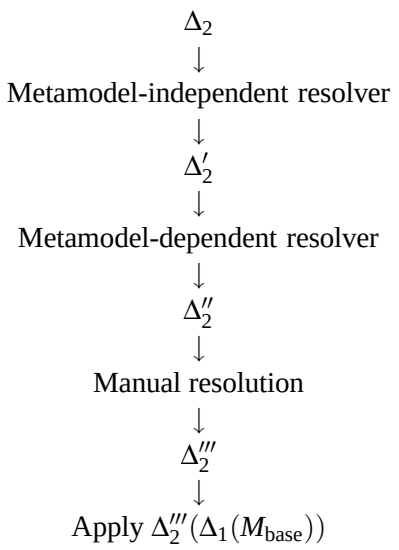


Figure 10: A complete merge system with three distinct resolution steps.

it back to the repository, merging their changes with the work of others.

It remains to be investigated if the above mechanism is feasible, or if a merging algorithm which would consider more than two differences in parallel is required [7] [10].

4.2 Representing Differences

Once we know how to calculate the differences between two models, we should consider how to store them in a repository. The XMI standard describes a system to represent differences in a model inside an XMI document (pp. 1-32 of [9]). According to the standard, a difference entity can be used to add, delete or replace an element in a model. Although this approach is valid, it is too coarse-grained. If we just want to represent that a class has changed its name, we need to replace the complete class in the model. We consider that the replace option in a difference entity should be specialized into basic operations that work at the property level instead of the element level.

We can represent these differences as follows. We assume that a difference represents a change of a property f of an element e with UUID u. Where necessary, there is another element e_t with UUID u_t. Depending on the type of the property, this might mean one of the following modifications:

- set(e, f, v_o, v_n) : Set the value of $e.f$ from v_o to v_n, for an attribute of primitive type.

- insert(e, f, e_t) : Add a link from $e.f$ to e_t, for an unordered property.

- remove(e, f, e_t) : Remove a link from $e.f$ to e_t, for an unordered property.
- insertAt(e, f, e_t, i) : Add a link from $e.f$ to e_t, at index i, for an ordered property.
- removeAt(e, f, e_t, i) : Remove a link from $e.f$ to e_t, which is at index i, for an ordered property.

These five difference operations at the property level should be complemented with two difference operations at the element level:

- add(e,t) : Create a new element of type t with the UUID of e. By default, a new element has all its properties set to their default values.
- del(e,t) : Delete an element of type t with the UUID of e. An element may only be deleted if all its properties are set to their default values.

By using finely-grained differences we reduce the disk space needed for the repository as well as increase its overall performance, since we've actually reduced the total amount of difference information. Also, in a distributed setting, network communication will be substantially faster as model differences are often significantly smaller than complete models.

Once we know that a model has changed, we need to know why it has changed. For this, additional metadata is required. While an informal description of the change goes a long way, formal, traceable reasons for the change are a boon to bring software engineering toward a robust scientific discipline. We would also like to keep the history of the model, and review several old versions of it.

4.3 Evolution of Elements

Quite often a new version of an element represents just an improvement from the previous version. But in many other cases, a new version of an element is derived from one or more other elements, possibly of a different type. This is the case when we e.g. create a new class that realizes the functionality described in a use case or create a statechart as a refinement of another statechart. A version control system keep tracks of edited elements but not of derived elements, nor the reason why they have been created.

Previous versions of the UML provided a model element named Flow to model evolution relationships. However, it seems that this element has been removed from UML 2.0. It was not supported by the main UML tools and in any case it was not useful to create traces between elements that resided in two different models or in models that were described in different modeling languages.

The long-term solution seems to be in yet another standard. The OMG has a request for proposals for a query, view and transformation language for MOF 2.0 (QVT) [8]. One of the operational requirements for the proposals is the ability to trace the execution of transformations. This can be achieved by defining a tuple

(S, T, r), where S and T represents sets of model elements and r a transformation rule such that $r(S) = T$. This can also be generalized to allow free-form editing of a model as a possible transformation.

Another requirement for the proposals is that they should provide a MOF-based metamodel for the proposed language. The actual metamodel varies from one proposal to another, but the important implication is that the MOF standard describes how to generate XMI documents from any MOF model. Once the standard is accepted, we will be able to represent the evolution history of a model as a sequence of transformation traces and we will be able to store the evolution history as a standard XMI document.

5 Access Control

Access control defines the mechanisms by which read or write access to parts of the model are defined and modified. For some types of projects, limiting access of developers to only some parts of the model is important, or even mandatory. As an example of limiting read access, security-related information is to be disclosed only to a specific set of developers. A more common scenario is limiting write access, such that a group of developers may work on a part of a model, and another group on another part.

It might feel intuitive to set the granularity of access at the element level, whereby read or write access is determined based on the elements that a developers wants to read or change. However, this may be impossible due to fact that associations in the metamodel are relations. Each metamodel association is represented as one property in each participating class. Modifying an association implies the modification of the two associated properties.

For example, consider a class which has write restrictions for some reason, perhaps due to being thoroughly tested. It would be impossible to create a new class as a subclass of this write-restricted class, since subclassing implies modifying the specialization property of that class—which the developer is not allowed to modify!

However, most developers would consider these changes as harmless to the original class. This is because while some properties carry semantic meaning for an element, other properties only act as a navigational aid or as the opposite end of a bidirectional meta-association.

Clearly, the level of detail in access control must be based on the properties of elements, not on the elements themselves. In some cases, the developer ought to be able to use a class, subclass it but not add new operations or change existing attributes. The distinction cannot be made by allowing or disallowing write access to the class element itself, but the properties of the class.

Finally, access control could also be coupled with intent. Project management might decide that every change to a model must include a reference to the task that it furthers, or a reference to a bug report that the change fixes. Part of access control could also imply verifying that the change actually does fix the bug, by building

the executable and running a previously defined unittest for it. Further examples where policy dictates access control abound; as an example, several open-source projects have an informal policy of discussing a program change and voting for or against before committing the change. The information relating these changes, review comments and votes could be stored in the repository as well.

6 Project Management Information

Software project management involves the definition, application and constant improvement of procedures and methods used in the development of a software product. A logical advancement in project management tools is to use metamodeling techniques to define, represent and manage the artifacts needed in project management.

From previous experience with source code, we know there are several kinds of project management data in which the designers are interested, and certainly this same data will be interesting for models as well.

Issue trackers manage the different kinds of errors that are found in a model. Furthermore they can keep track of items that need to be implemented. An interesting consequence of using models here is that generating unittests for errors might partially be automated. Also, scripts for issue tracking could connect with estimation models which could take more information into account from the issue tracker. Estimates can include time as well as other resources such as available personnel, money, hardware, production facilities or logistics.

Testing frameworks take care of running unittests, calculating test coverage in the model, and running acceptance tests. Again, quality assurance teams will benefit from using models, since it lowers the barrier between the tests and the issue tracker; modeling unifies the namespace by standardizing query and management of data.

Finally, workflow definitions define how the process of developing is to be carried out. Everything from loosely specified guidelines where anything is allowed to meticulously strict development processes can be modeled. Using models might imply that there is little need to create separate parsers for all kinds of data, and thus managing and querying the existing or generated data can be made possible in the first place, perhaps even automated. Synchronization with all the other facilities of the project management system is as casual as any other modeling activity, and workflow can easily be enforced by a complete project management system consisting solely of models.

7 Conclusions

In this article we have reviewed the basic requirements for a model-based configuration system. A true model-based repository is an essential element in any software development project where models are created and updated constantly. It

is also a key element in an MDA-based project [3], where multiple versions of the same model are created simultaneously in order to separate the problem space from the implementation concerns.

The construction of such a system may not seem an issue. There exists already many similar systems to manage source code and XML documents. XMI, the standard model interchange format, is based on XML so it may seem that we can simply use any of the existing systems. However, in this article we have seen that XML tools may be too generic and that there are many open issues not addressed by the standards.

In one way, XMI is a large step forward to ensure basic interoperability between UML tools. It also allows the development of simple tools to extract information and transform UML models easily [12, 11]. However, the standard should be improved. XMI needs a better mechanism to identify global model elements, including elements from the standard libraries of programming languages. Also, the XMI difference entities are far from optimal and should be refined into individual updates of properties. In the meanwhile, OMG standards have so far concentrated heavily on the structural aspects of modeling and metamodeling. Dynamic aspects such as transformation, access control, policy management, element versioning and evolution, model transportation and seamless language evolution are still not defined in the context of UML and MDA.

References

[1] Marcus Alanen. A Meta Object Facility-Based Model Repository With Version Capability, Optimistic Locking and Conflict Resolution. Master's thesis, Åbo Akademi University, November 2002.

[2] Marcus Alanen and Ivan Porres. Difference and Union of Models. In *Proceedings of the UML 2003 Conference*, October 2003.

[3] OMG Architecture Board. Model Driven Architecture - A Technical Perspective. OMG Document ormsc/01-07-01. Available at www.omg.org, 2001.

[4] CAE Specification. DCE 1.1: Remote Procedure Call, 1997. Available at `http://www.opengroup.org/onlinepubs/9629399/toc.htm`.

[5] G. Clemm, J. Amsden, T. Ellison, C. Kaler, and J. Whitehead. Versioning Extensions to WebDAV, RFC 3253, March 2002. Available at `http://www.ietf.org/rfc/rfc3253.txt`.

[6] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. HTTP Extensions for Distributed Authoring — WEBDAV, RFC 2518, February 1999. Available at `http://www.ietf.org/rfc/rfc2518.txt`.

[7] Tom Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.

[8] OMG. MOF 2.0 Query / Views / Transformations RFP. OMG Document ad/02-04-10. Available at www.omg.org, 2002.

[9] OMG. OMG XML Metadata Interchange (XMI) Specification. OMG Document 03-05-02. Available at www.omg.org, 2003.

[10] Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. Parallel Changes in Large Scale Software Development: An Observational Case Study. In *Proceedings of the International Software Engineering Conference*, April 1998.

[11] I. Porres. A Toolkit for Manipulating UML Models. Technical Report 441, TUCS Turku Centre for Computer Science, 2002. Available at www.tucs.fi.

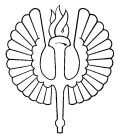
[12] P. Stevens. Small-scale XMI programming: a revolution in UML tool use? *Automated Software Engineering*, 10(1):7–21, January 2003.

[13] W3C. XQuery 1.0: An XML Query Language (working draft). Available at `http://www.w3.org/TR/xquery/`., August 2003.

[14] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.fi



University of Turku
- **Department of Information Technology**
- **Department of Mathematics**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**